

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ  
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий  
Кафедра Программной инженерии  
Специальность 1-40 01 01 «Программное обеспечение информационных технологий»  
Специализация 1-40 01 01 10 «Программное обеспечение информационных технологий  
(программирование интернет-приложений)»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  
к курсовому проекту на тему:**

Web-приложение для подбора сотрудников и поиска работы

Выполнил студент Орлов Алексей Олегович  
(Ф.И.О.)

Руководитель проекта к.т.н., доцент Смелов В.В.  
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доцент Смелов В.В.  
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой \_\_\_\_\_

## Содержание

Содержание.....	2
Введение .....	4
1 Постановка задачи и обзор аналогичных решений .....	5
1.1 Постановка задачи.....	5
1.2 Аналитический разбор аналогов .....	5
1.2.1 Облачный мессенджер для удобного и безопасного общения «Telegram Web» .....	5
1.2.2 Платформа для общения и совместной работы «teams.live.com».....	6
1.3 Выводы по разделу.....	8
2 Проектирование web-приложения .....	9
2.1 Функциональность web-приложения.....	9
2.2 Структура базы данных.....	12
2.3 Архитектура web-приложения .....	14
2.4 Выводы по разделу .....	15
3 Реализация web-приложения .....	17
3.1 Программная платформа Node.js .....	17
3.2 Реляционная база данных PostgreSQL .....	17
3.3 Средство моделирования объектной структуры данных.....	17
3.4 Программные библиотеки.....	21
3.5 Структура серверной части.....	23
3.6 Реализация функций для абонента.....	24
3.6.1 Регистрация .....	24
3.6.2 Аутентификация .....	25
3.6.3 Добавление пользователя в контакт .....	25
3.6.4 Просмотр списка контактов.....	26
3.6.5 Удаление контакта.....	26
3.6.6 Отправление и получение сообщения .....	27
3.6.7 Редактирование и удаление сообщений .....	28
3.6.8 Просмотр списка каналов .....	28
3.6.9 Просмотр и изменение коллекции настроек .....	29
3.7 Реализация функций для администратора .....	30
3.7.1 Аутентификация .....	30
3.7.2 Просмотр списка пользователей .....	30
3.7.3 Блокировать и разблокировать пользователей .....	31
3.7.4 Добавление, удаление и редактирование канала.....	31
3.8 Реализация функций для главного администратора .....	32
3.8.1 Аутентификация .....	32
3.8.2 Просмотр списка всех абонентов и администраторов .....	32

3.8.3 Удаление абонентов .....	33
3.8.4 Изменение ролей пользователей .....	33
3.8.5 Просмотр наиболее активных пользователей, общего количества пользователей .....	34
3.9 Структура клиентской части.....	34
3.11 Выводы по разделу .....	35
4 Тестирования web-приложения.....	37
4.1 Функциональное тестирование .....	37
4.2 Выводы по разделу .....	40
5 Руководство пользователя .....	41
5.1 Руководство абонента.....	41
5.1.1 Регистрация .....	41
5.1.1 Аутентификация .....	41
5.2 Руководство администратора .....	42
5.2.1 Добавление канала.....	42
5.2.2 Изменение канала .....	44
5.2.3 Удаление канала.....	44
5.2.4 Блокирование пользователя.....	45
5.3 Руководство главного администратора .....	46
5.3.1 Просмотр пользователей и администраторов, изменение роли пользователя .....	46
5.3.2 Удаление пользователя .....	47
Заключение .....	48
Список используемых источников.....	49
Приложение А .....	50
Приложение Б.....	51
Приложение В .....	55
Приложение Г .....	68

## Введение

Web-приложение мессенджера – это программное обеспечение, предназначенное для организации удобного и безопасного общения между абонентами. Оно предоставляет инструменты для обмена текстовыми сообщениями, файлами и видеозвонками, а также функциональность создания групповых чатов.

Цели проекта:

- повысить эффективность и скорость внутренней и внешней коммуникации пользователей;
- предоставить защищенный и стабильный канал связи для частного и делового общения;
- обеспечить удобный интерфейс для общения.

Для достижения поставленной цели были определены следующие задачи:

- проанализировать существующие платформы для поиска работы, с рассмотрением их сильных и слабых сторон. Аналоги описаны в главе 1.
- разработать архитектуру и структуру web-приложения, обосновать выбор технологий и спроектировать базу данных. Проектирование описано в главе 2.
- реализовать приложение, включая разработку ключевых функций. Реализация описана в главе 3.
- провести тестирование web-приложения. Тестирование описано в главе 4.
- описать руководство пользователя. Руководство пользователя описано в главе 5.

Целевая аудитория — это частные пользователи и компании, нуждающиеся в удобном, безопасном и функциональном средстве для личной и корпоративной переписки.

Техническая реализация проекта основана на асинхронном программировании с использованием языка TypeScript и платформы Node.js [1]. Программное средство взаимодействует с базой данных и может быть запущено на различных платформах.

## **1 Постановка задачи и обзор аналогичных решений**

### **1.1 Постановка задачи**

В Web-приложении подразумевается несколько ролей для различных пользователей, такие как абонент, администратор и главный администратор.

Абонент должен иметь возможность зарегистрироваться и аутентифицироваться в системе, искать и добавлять пользователей в контакт, просматривать список контактов, удалять контакты. Начать видеозвонок со своим контактом. Отправлять и получать сообщения и файлы, просматривать, редактировать и удалять сообщения. Просматривать список каналов. Просматривать и изменять коллекцию настроек.

Администратор должен иметь возможность аутентифицироваться в системе, просматривать список абонентов, заблокировать и разблокировать их. Добавить, редактировать и удалить канал.

Главный администратор должен иметь возможность аутентифицироваться в системе, просматривать список абонентов и администраторов. Удалять абонентов, изменять им роль, просматривать наиболее активных абонентов, а также просматривать их общее количество. Экспортировать данные в JSON и импортировать данные из JSON.

### **1.2 Аналитический разбор аналогов**

В этом разделе будут приведены web-приложения существующих мессенджеров и платформ для общения и совместной работы.

#### **1.2.1 Облачный мессенджер для удобного и безопасного общения «Telegram Web»**

На рисунке 1.1 представлен скриншот интерфейса на «*web.telegram.com*» [3].

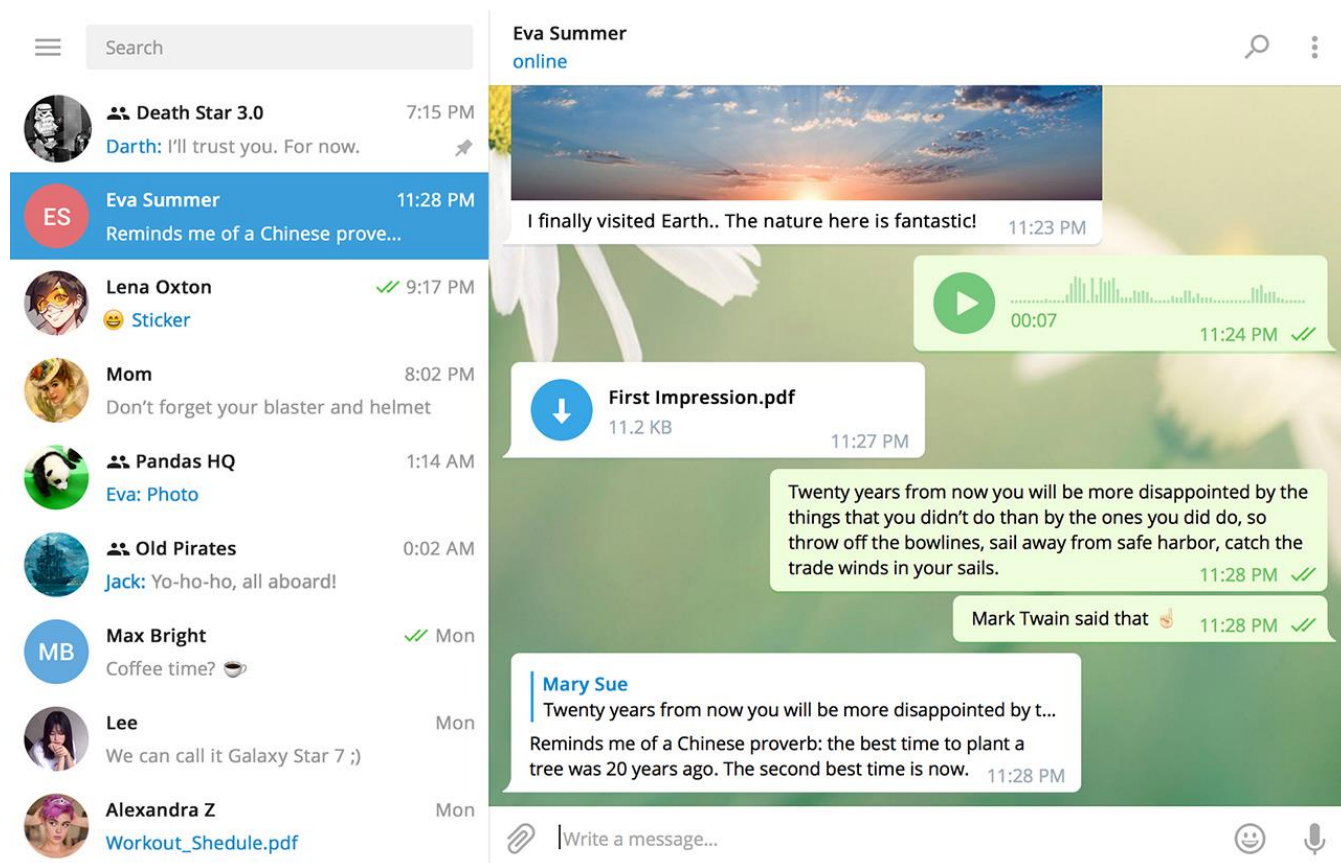


Рисунок 1.1 – Интерфейс на сайте «*web.telegram.com*»

Преимущества платформы:

- быстрый доступ и кроссплатформенность: доступ к переписке и функциям Telegram Web прямо из браузера без установки приложений — удобно для работы с любого устройства;

- широкая сеть общения: миллионы пользователей по всему миру — возможность вести как личную, так и деловую переписку, участвовать в группах и подписываться на каналы;

- инструменты для автоматизации: поддержка ботов, интеграций и уведомлений делает Telegram Web полезным для бизнеса, технической поддержки и оповещений.

Недостатки Telegram Web:

- ограниченная функциональность по сравнению с мобильной версией: отсутствие звонков, некоторых настроек и функций (например, секретных чатов);

- отсутствие расширенной фильтрации чатов и контента: сложнее находить нужную информацию в больших группах и каналах.

Telegram Web — это удобный инструмент для мгновенной переписки и получения информации, особенно полезный в рабочих и повседневных задачах. Однако веб-версия уступает мобильным приложениям в полноте функционала и требует постоянного подключения к сети.

### 1.2.2 Платформа для общения и совместной работы «*teams.live.com*»

На рисунке 1.2 представлен интерфейс на платформе «teams.live.com»[2].

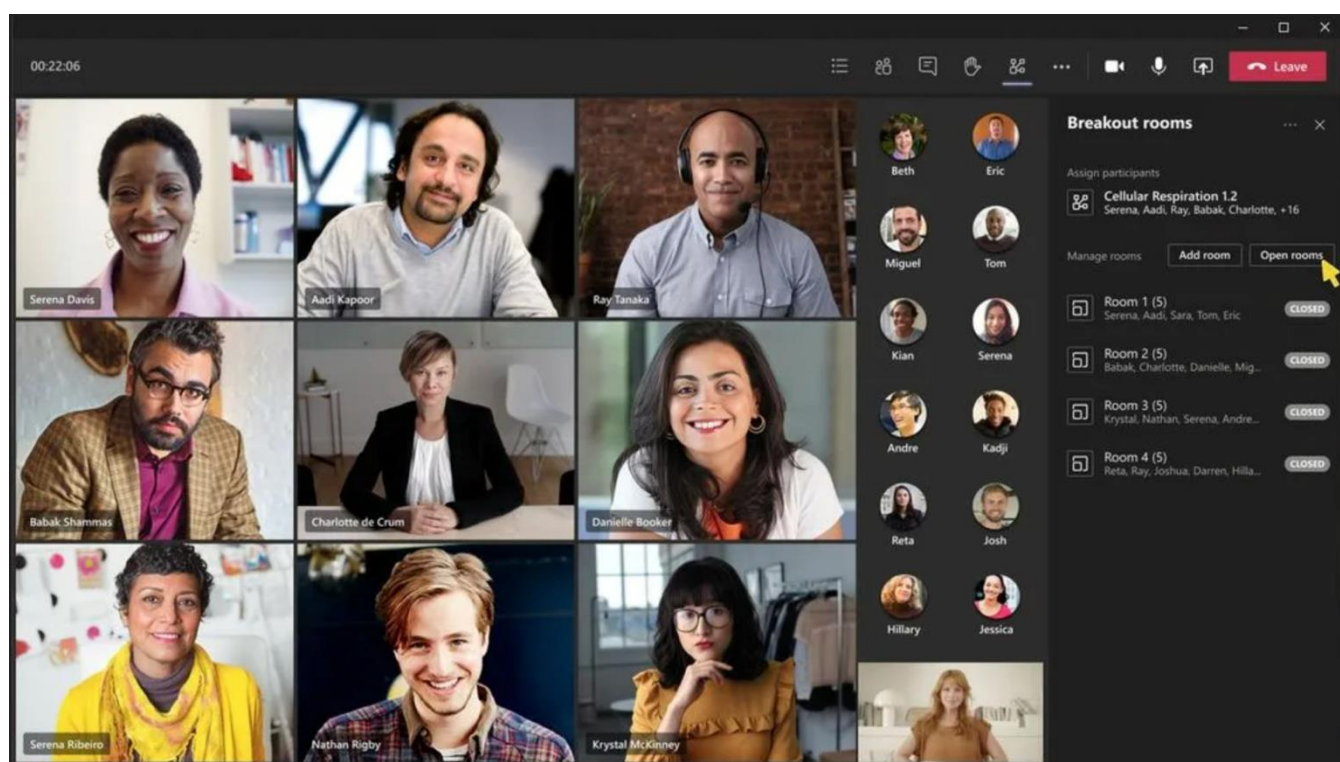


Рисунок 1.2 – Интерфейс на «teams.live.com»

Преимущества платформы «Microsoft Teams»:

- удобная организация общения: структурированные чаты, каналы и команды позволяют вести переписку и совместную работу по проектам без путаницы;
- функциональные видеозвонки и встречи: встроенные средства для онлайн-конференций, расписаний и совместных рабочих сессий;
- личный рабочий кабинет: индивидуальные настройки уведомлений, управление задачами, чатами и файлами — все в одном месте.

Недостатки платформы «Microsoft Teams»:

- сложность освоения для новых пользователей: интерфейс может показаться перегруженным, особенно при первом использовании;
- высокие требования к системе и интернету: особенно при видеозвонках или одновременной работе с несколькими документами;
- несмотря на широкие возможности для корпоративного использования, платформа лучше всего работает в экосистеме Microsoft (Office 365, OneDrive, SharePoint), а подключение некоторых внешних инструментов требует дополнительных настроек или подписки;
- по сравнению с мессенджерами вроде Telegram, в Teams меньше возможностей для персонализации чатов.

Microsoft Teams — это мощная платформа для корпоративной коммуникации и совместной работы, особенно эффективная в рамках экосистемы Microsoft. Однако она ориентирована в первую очередь на внутренние бизнес-процессы и требует времени на адаптацию новых пользователей.

### 1.3 Выводы по разделу

Анализ аналогичных решений показал, что существующие web-платформы для общения, такие как «Telegram Web» и «Microsoft Teams», обладают широкими возможностями для взаимодействия между абонентами. «Telegram Web» выделяется своей скоростью, простотой интерфейса и поддержкой ботов, но часть продвинутой функциональности (например, звонки и секретные чаты) недоступна в веб-версии. «Microsoft Teams», в свою очередь, ориентирован на корпоративную аудиторию и предоставляет мощные инструменты для совместной работы, однако требует платной подписки и обладает более сложным интерфейсом, что затрудняет освоение для новых пользователей.

Поставленные задачи требуют разработки Web-приложения с поддержкой трех ролей: абонента, администратора и главного администратора. Гость должен иметь возможность зарегистрироваться и аутентифицироваться в системе. Администратору необходимо предоставить возможность аутентификации, доступ к инструментам модерации, управления абонентами и каналами. Главный администратор в свою очередь должен иметь доступ к полноценным инструментам администрирования, наделять абонентов правами администратора, а также снимать их при необходимости, просматривать общее количество всех абонентов, а также следить за их активностью.

Кроме того, важно предусмотреть масштабируемость системы, чтобы при увеличении числа пользователей не возникало проблем с производительностью. Для этого можно использовать микросервисную архитектуру, которая позволит распределить нагрузку между разными серверами.

Еще одним ключевым аспектом является удобство интерфейса. Платформа должна быть интуитивно понятной как для обычных пользователей, так и для администраторов. Важно продумать адаптивный дизайн, чтобы приложение корректно отображалось на любых устройствах – от смартфонов до десктопов. Дополнительно можно реализовать гибкие настройки интерфейса, позволяющие пользователям персонализировать внешний вид чатов и панелей управления под свои предпочтения.



## 2 Проектирование web-приложения

### 2.1 Функциональность web-приложения

Функциональность web-приложения представлена на диаграмме вариантов использования. Диаграмма вариантов использования представлена на рисунке 2.1.

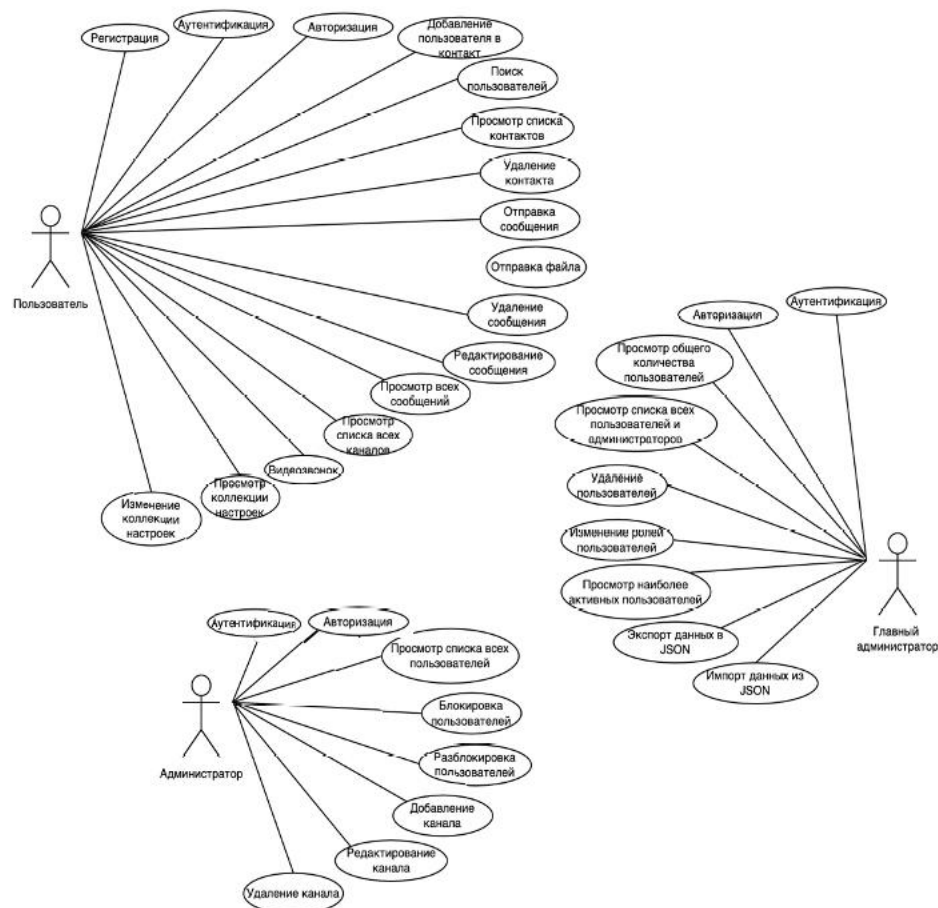


Рисунок 2.1 – Диаграмма вариантов использования

Диаграмма вариантов использования наглядно демонстрирует доступные функции для каждой роли и способы их взаимодействия с приложением. Список ролей и их описание представлены в таблице 2.1.

Таблица 2.1 – Роли пользователей *web*-приложения

Роль	Назначение
Абонент	Пользователь с базовой ролью, получающей доступ ко всем основным функциям мессенджера после регистрации и аутентификации. Абонент может находить других пользователей, добавлять их в контакты, общаться в личных чатах и каналах, отправлять и получать файлы, совершать видеозвонки, а также настраивать параметры своего профиля.

Окончание таблицы 2.1

Администратор	Пользователь с расширенными правами управления контентом и платформы. Администратор может управлять публичными каналами (создание, редактирование, удаление), а также осуществлять модерацию: просматривать список пользователей, блокировать и разблокировать их. Основная цель администратора — поддержание порядка и контроль за контентом внутри мессенджера.
Главный администратор	Пользователь с наивысшими правами доступа. Может управлять всеми пользователями и администраторами, изменять их роли, удалять аккаунты, просматривать статистику активности и общее количество пользователей. Также обладает возможностью импорта и экспорта данных в формате JSON. Основная цель главного администратора — контроль всей платформы, управление доступом и анализ пользовательской активности.

Описание функциональности приложения, доступной для роли абонента представлено в таблице 2.2.

Таблица 2.2 – Функциональность для роли абонента

Номер функции	Название функции	Пояснение
1	Регистрация	Процесс регистрации в системе с использованием учетных данных, таких как email, имя пользователя, пароль.
2	Аутентификация	Процесс входа в систему с использованием учетных данных, таких как email и пароль.
3	Добавление пользователя в контакт	Функция для добавления пользователя в контакты.
4	Поиск пользователей	Функция для поиска пользователей.
5	Просмотр списка контактов	Функция для просмотра списка контактов.
6	Удаление контакта	Функция для удаления контакта.
7	Отправка сообщения	Функция для отправки сообщения.
8	Отправка файла	Функция для отправки файла в сообщении.
9	Удаление сообщения	Функция для удаления сообщения.
10	Редактирование сообщения	Функция для редактирования сообщения.
11	Просмотр всех сообщений	Функция для просмотра всех сообщений.

Окончание таблицы 2.2

12	Просмотр списка всех каналов	Функция для просмотра списка всех каналов.
13	Видеозвонок	Функция для осуществления видеозвонка.
14	Просмотр коллекции настроек	Функция для просмотра коллекции настроек.
15	Изменение коллекции настроек	Функция для изменения коллекции настроек.

Описание функциональности приложения, доступной для роли администратора представлено в таблице 2.3.

Таблица 2.3 – Функциональность для роли администратора

Номер функции	Название функции	Пояснение
1	Аутентификация	Процесс входа в систему с использованием учетных данных, таких как email и пароль.
2	Просмотр списка всех пользователей	Функция для просмотра списка всех пользователей.
3	Блокировка пользователей	Функция для блокирования пользователей.
4	Разблокировка пользователей	Функция для снятия блокировки с пользователя.
5	Добавление канала	Функция для добавления канала.
6	Редактирование канала	Функция для редактирования канала.
7	Удаление канала	Функция для удаления канала.

Описание функциональности приложения, доступной для роли главного администратора представлено в таблице 2.4.

Таблица 2.4 – Функциональность для роли главного администратора

Номер функции	Название функции	Пояснение
1	Аутентификация	Процесс входа в систему с использованием учетных данных, таких как email и пароль.
2	Просмотр общего количества пользователей	Функция для просмотра общего количества пользователей.
3	Просмотр списка всех пользователей и администраторов	Функция для просмотра списка всех пользователей и администраторов.
4	Удаление пользователей	Функция для удаления пользователей.

Окончание таблицы 2.4

5	Изменение роли пользователей	Функция для изменения роли пользователя.
6	Просмотр наиболее активных пользователей	Функция для просмотра наиболее активных пользователей по количеству сообщений.
7	Экспорт данных в JSON	Функция для экспорта данных.
8	Импорт данных из JSON	Функция для импорта данных.

В рамках проектирования web-приложения, функциональность была детализирована на основе диаграммы вариантов использования и ролей пользователей. Каждая роль в системе (абонент, администратор и главный администратор) имеет уникальные задачи и функции, доступные только им.

## 2.2 Структура базы данных

Для хранения данных о пользователях и сообщений, была выбрана база данных *Postgres 16.4*[4].

Логическая схема базы данных представлена на рисунке 2.2.

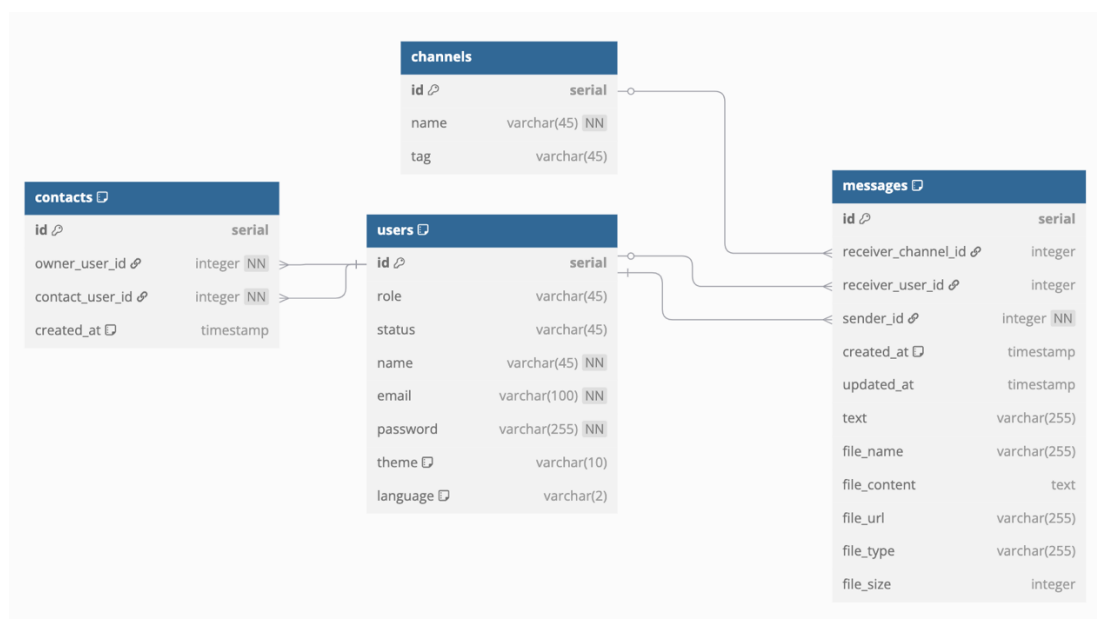


Рисунок 2.2 – Логическая схема базы данных

Схема базы данных разрабатываемого приложения состоит из 4 таблиц, состоящих из различных полей и столбцов. Были созданы следующие коллекции: Users, Channels, Messages, Contacts. Наименования и описание всех сущностей базы данных приведены в таблице 2.5.

Таблица 2.5 – Таблицы базы данных

Название таблицы	Назначение коллекции
Users	Хранение данных о пользователе.
Channels	Хранение данных о канале.
Messages	Хранение данных о сообщении.
Contacts	Хранение данных о контакте.

Далее подробнее про каждую из таблиц базы данных.

Таблица Users хранит данные о пользователях. Состоит из полей, представленных в таблице 2.6.

Таблица 2.6 – Поля таблицы Users

Объект	Тип данных	Описание
id	serial	Уникальный идентификатор пользователя.
role	varchar(45)	Роль пользователя.
status	varchar(45)	Статус пользователя.
name	varchar(45)	Имя пользователя.
email	varchar(100)	Почта пользователя.
password	varchar(64)	Хеш пароля пользователя.
theme	string	Тема интерфейса пользователя.
language	string	Язык интерфейса пользователя.

Таблицы Channels хранит данные о публичных каналах. Состоит из полей, представленных в таблице 2.7.

Таблица 2.7 – Поля таблицы Channels

Объект	Тип данных	Описание
id	serial	Уникальный идентификатор канала.
name	varchar(45)	Имя канала.
tag	varchar(45)	Тег канала.

Таблица Messages хранит данные о сообщениях. Состоит из полей, представленных в таблице 2.8.

Таблица 2.8 – Поля таблицы Messages

Объект	Тип данных	Описание
id	serial	Уникальный идентификатор сообщения.
sender_id	integer	Уникальный идентификатор пользователя, который отправляет сообщение.
receiver_channel_id	integer	Уникальный идентификатор канала, на который пользователь отправляет сообщение.

Окончание таблицы 2.8

receiver_user_id	integer	Уникальный идентификатор пользователя, получающего сообщение от другого пользователя.
created_at	timestamp	Дата создания сообщения. Автоматически заполняется при создании сообщения.
updated_at	timestamp	Дата изменения сообщения. Автоматически обновляется при редактировании сообщения.
text	varchar(255)	Текст сообщения.
file_name	varchar(255)	Имя файла, прикрепленного к сообщению.
file_content	text	Контент файла, прикрепленного к сообщению.
file_url	varchar(255)	Путь к файлам на сервере, из которого выгружаются статические файлы.
file_type	varchar(255)	Тип файла, прикрепленного к сообщению.
file_size	integer	Размер файла, прикрепленного к сообщению.

Таблицы Contacts хранит данные о контактах. Состоит из полей, представленных в таблице 2.9.

Таблица 2.9 – Поля таблицы Contacts

Объект	Тип данных	Описание
id	serial	Уникальный идентификатор контакта.
owner_user_id	integer	Уникальный идентификатор пользователя, который добавил контакт.
contact_user_id	integer	Уникальный идентификатор пользователя, который добавлен в контакты владельцем.
created_at	timestamp	Дата создания контакта. Автоматически заполняется при создании контакта.

Каждая таблица имеет четко определенные поля, отражающие определенные аспекты работы платформы для общения. Определены типы данных полей и связи между таблицами.

## 2.3 Архитектура web-приложения

Архитектура web-приложения представлена на рисунке 2.3.

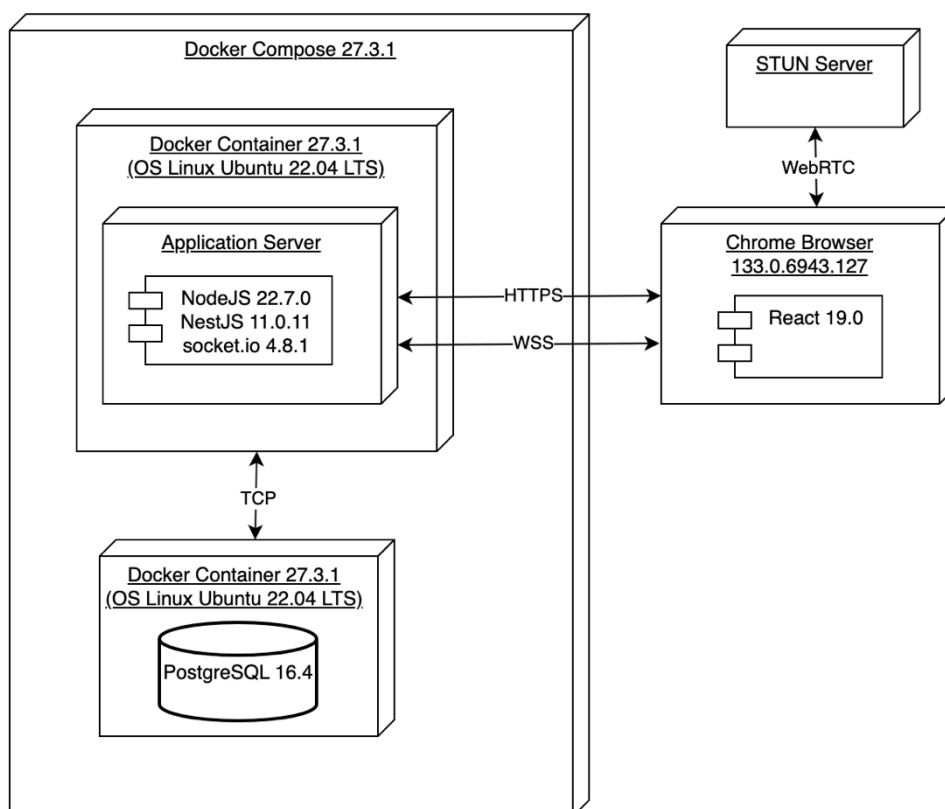


Рисунок 2.3 – Архитектура web-приложения

Приложение разворачивается с помощью docker-compose в 3 контейнерах docker. Пользователь взаимодействует с web-приложением через web-браузер.

Пояснение назначения каждого элемента web-приложения представлено в таблице 2.11.

Таблица 2.11 – Назначение элементов web-приложения

Элемент	Назначение
Database Server (Postgres)	Используется для хранения и предоставления доступа к данным, которые необходимы для работы web-приложения.
Application Server	Обрабатывать запросы пользователя, запрашивать данные из базы данных
Brave Browser	Отображать фронтенд-часть web-приложения, отправлять запросы пользователя, отображать ответы сервера.

Таким образом были рассмотрены все ключевые элементы архитектуры web-приложения.

## 2.4 Выводы по разделу

В данном разделе была подробно рассмотрена функциональность разрабатываемого web-приложения с поддержкой трех ролей пользователей: абонента, администратора и главного администратора. Абонент может зарегистрироваться в системе, выполнять почти весь функционал мессенджера. Администратору доступен функционал для осуществления модерации в web-приложении, создавать, изменять и удалять каналы, блокировать и разблокировать пользователей. Главный администратор осуществляет полное администрирование в web-приложении, способен менять роли пользователей, просматривать всех пользователей, их количество и наиболее активных пользователей.

Логическая схема базы данных построена с использованием Postgres и включает четыре основные таблицы: Users, Channels, Messages и Contacts. Каждая таблица содержит четко структурированные данные, отражающие ключевые особенности и процессы работы платформы: регистрацию пользователей, добавление контактов и каналов, отправку текстовых сообщений и файлов, видеозвонки. Схема отличается строгой типизацией, оптимальными индексами для ускорения запросов и каскадными ограничениями для целостности данных, поддерживая партиционирование и репликацию для масштабируемости, а также включает механизмы шифрования чувствительных данных и ролевую модель доступа для безопасности, что позволяет платформе эффективно обрабатывать регистрацию пользователей, обмен сообщениями, видеозвонки и групповые взаимодействия при высокой нагрузке с возможностью дальнейшего расширения функционала.

Также была представлена архитектура web-приложения, реализованная с использованием Docker и состоящая из двух контейнеров: Postgres для хранения данных и backend-сервера, обрабатывающего пользовательские запросы. Вся система взаимодействует с пользователем через web-браузер.



## 3 Реализация web-приложения

### 3.1 Программная платформа Node.js

Для разработки серверной части проекта была выбрана платформа Node.js, известная своей неблокирующей моделью ввода-вывода и однопоточным выполнением, что делает ее особенно подходящей для высоконагруженных приложений. В качестве основного фреймворка используется NestJS [5], обеспечивающий удобную и модульную архитектуру для построения масштабируемых серверных приложений. NestJS предоставляет мощный набор инструментов для маршрутизации, обработки HTTP-запросов, работы с базами данных и внедрения зависимостей. Внутри NestJS используется Express, как основной HTTP-сервер, что позволяет совмещать гибкость Express с структурированностью NestJS. Листинг кода модуля App представлен в приложении А.

### 3.2 Реляционная база данных PostgreSQL

В качестве системы управления базами данных для проект используется PostgreSQL — мощная объектно-реляционная СУБД с открытым исходным кодом. PostgreSQL поддерживает строго типизированные таблицы, транзакции, внешние ключи и полнотекстовый поиск, что ее надежным решением для хранения структурированных данных. Она обладает высокой степенью расширяемости, поддерживает хранимые процедуры, триггеры и сложные SQL-запросы. Также PostgreSQL обеспечивает масштабируемость и отказоустойчивость за счет встроенной поддержки репликации и кластеризации.

### 3.3 Средство моделирования объектной структуры данных

В проекте используется средство объектно-реляционного отображения TypeORM[7], которое предоставляет удобный способ работы с реляционными базами данных, включая PostgreSQL. TypeORM позволяет описывать структуру таблиц базы данных в виде Entity-классов, обеспечивая строгую типизацию, контроль связей и автоматическую генерацию миграций. Поддерживает выполнение всех необходимых операций с базой данных (CRUD) и легко интегрируется с архитектурой NestJS.

Сущность User, описывающая таблицу Users представлена на листинге 3.1:

```
@Unique(['email'])
@Entity('users')
export class User {
  @PrimaryGeneratedColumn()
  id: number;
  @Column('varchar', { length: 45 })
  role: string;
  @Column('varchar', { length: 45 })
  status: string;
```

```

@Column('varchar', { length: 45 })
name: string;
@Column('varchar', { length: 100 })
email: string;
@Column('varchar', { length: 255, select: false })
password: string;
@Column('varchar', { length: 10 })
theme: string;
@Column('varchar', { length: 2 })
language: string;
@OneToMany(() => Contact, (contact) => contact.owner)
contactsAsOwner: Contact[];
@OneToMany(() => Contact, (contact) => contact.contact)
contactsAsContact: Contact[];
@OneToMany(() => Message, (message) => message.sender)
sentMessages: Message[];
@OneToMany(() => Message, (message) => message.receiverUser)
receivedMessages: Message[];
messageCount?: number;
}

```

Листинг 3.1 – Сущность User

Сущность User описывает структуру таблицы Users полями, представленными в таблице 3.1:

Таблица 3.1 – Поля сущности User

Объект	Тип данных	Описание
id	number	Уникальный идентификатор пользователя.
role	string	Роль пользователя.
status	string	Статус пользователя.
name	string	Имя пользователя.
email	string	Электронная почта пользователя (unique).
password	string	Роль пользователя в системе.
theme	string	Тема интерфейса пользователя.
language	string	Язык интерфейса пользователя.
messageCount	number	Количество отправленных сообщений пользователем.

Сущность Channels, описывающая таблицу Channels представлена на листинге 3.2:

```

@Entity('channels')
export class Channel {
  @PrimaryGeneratedColumn()
  id: number;
  @Column('varchar', { length: 45 })
  name: string;
}

```

```

@Column('varchar', { length: 45 })
tag: string;
@OneToMany(() => Message, (message) => message.receiverChannel)
messages: Message[];
}

```

### Листинг 3.2 – Сущность Channels

Сущность Channels описывает структуру таблицы Channels полями, представленными в таблице 3.2:

Таблица 3.2 – Поля сущности Channels

Объект	Тип данных	Описание
id	number	Уникальный идентификатор канала.
name	string	Имя канала.
tag	string	Тег канала.

Сущность Messages, описывает структуру таблицы Messages полями, представленными на листинге 3.3:

```

@Entity('messages')
export class Message {
  @PrimaryGeneratedColumn()
  id: number;
  @ManyToOne(() => User, (user) => user.sentMessages, {nullable:
false, onDelete: 'SET NULL',})
  @JoinColumn({ name: 'sender_id' })
  sender: User;
  @ManyToOne(() => Channel, (channel) => channel.messages,
{onDelete: 'CASCADE',})
  @JoinColumn({ name: 'receiver_channel_id' })
  receiverChannel: Channel;
  @ManyToOne(() => User, (user) => user.receivedMessages, {onDelete:
'CASCADE',})
  @JoinColumn({ name: 'receiver_user_id' })
  receiverUser: User;
  @CreateDateColumn({type: 'timestamp', default: () =>
'CURRENT_TIMESTAMP',})
  created_at: Date;
  @UpdateDateColumn({type: 'timestamp', nullable: true,})
  updated_at: Date;
  @Column({type: 'varchar', length: 255, nullable: true,})
  text: string;
  @Column({name: 'file_name', type: 'varchar', length: 255, nullable:
true,})
  fileName: string;
  @Column({name: 'file_content', type: 'text', nullable: true,})
  fileContent: string;
  @Column({name: 'file_url', type: 'varchar', length: 255, nullable:
true,})
  fileUrl: string;
}

```

```

    @Column({name: 'file_type', type: 'varchar', length: 255, nullable:
true, })
    fileType: string;
    @Column({name: 'file_size', type: 'integer', nullable: true, })
    fileSize: number;
}

```

### Листинг 3.3 – Сущность Messages

Сущность Messages описывает структуру таблицы Messages полями, представленными в таблице 3.3:

Таблица 3.3 – Поля сущности Messages

Объект	Тип данных	Описание
id	number	Уникальный идентификатор сообщения.
created_at	Date	Дата создания сообщения.
updated_at	Date	Дата обновления сообщения.
text	string	Текст сообщения.
file_name	string	Имя файла в сообщении.
file_content	string	Контент файла в сообщении.
file_url	string	Путь к файлам на сервере.
file_type	string	Теги для поиска.
file_size	number	Размер файла в сообщении.

Сущность Contacts, описывает структуру таблицы Contacts полями, представленными на листинге 3.4:

```

@Entity('contacts')
export class Contact {
    @PrimaryGeneratedColumn()
    id: number;
    @ManyToOne(() => User, (user) => user.contactsAsOwner,
        onDelete: 'CASCADE',
    )
    owner: User;
    @JoinColumn({ name: 'owner_user_id' })
    @ManyToOne(() => User, (user) => user.contactsAsContact,
        onDelete: 'CASCADE',
    )
    contact: User;
    @CreateDateColumn({
        type: 'timestamp',
        default: () => 'CURRENT_TIMESTAMP',
    })
    created_at: Date;
}

```

### Листинг 3.4 – Сущность Contacts

Сущность `Contacts` описывает структуру таблицы `Contacts` полями, представленными в таблице 3.4:

Таблица 3.4 – Поля сущности `Contacts`

Объект	Тип данных	Описание
<code>id</code>	<code>number</code>	Уникальный идентификатор контакта.
<code>created_at</code>	<code>Date</code>	Дата создания контакта.

### 3.4 Программные библиотеки

В процессе разработки серверной части web-приложения для обеспечения её функциональности и повышения эффективности работы системы были использованы программные библиотеки, представленные в таблице 3.5.

Таблица 3.5 – Программные библиотеки серверной части

Библиотека	Версия	Назначение
<code>axios</code>	1.9.0	Библиотека для выполнения HTTP-запросов с клиента или сервера.
<code>argon2</code>	0.41.1	Библиотека для хеширования паролей с использованием алгоритма Argon2.
<code>bcryptjs</code>	3.0.2	Альтернатива <code>bcrypt</code> , реализованная на чистом JavaScript.
<code>class-transformer</code>	0.5.1	Библиотека для преобразования объектов между классами. Полезно для DTO.
<code>class-validator</code>	0.14.1	Валидатор данных. Используется вместе с DTO для проверки входных данных.
<code>jsonwebtoken</code>	9.0.2	Библиотека для работы с JWT (JSON Web Token).
<code>passport-local</code>	1.0.0	Стратегия Passport [11] для аутентификации по логину и паролю.
<code>passport</code>	0.7.0	Middleware для аутентификации в приложениях Node.js.
<code>passport-jwt</code>	4.0.1	Стратегия для Passport, позволяющая использовать JWT для аутентификации.
<code>pg</code>	8.14.1	Официальный клиент PostgreSQL для Node.js. Подключается к базе данных Postgres.
<code>postgres</code>	3.5.5	Минималистичный альтернативный драйвер для PostgreSQL.
<code>reflect-metadata</code>	0.2.2	Библиотека для работы с метаданными, используемая в NestJS и TypeScript.

Окончание таблицы 3.5

rxjs	7.8.1	Библиотека для реактивного программирования с использованием Observable.
socket.io	4.8.1	Библиотека для работы с WebSocket-соединениями (реального времени).
typeorm	0.3.21	Современный ORM для TypeScript/JavaScript. Используется с PostgreSQL и другими СУБД.

В процессе разработки клиентской части web-приложения были задействованы программные библиотеки, представленные в таблице 3.6.

Таблица 3.6 – Программные библиотеки клиентской части

Библиотека	Версия	Назначение
@babel/preset-react	7.27.1	Пресет Babel для компиляции JSX и других возможностей React. Используется при сборке проекта.
axios	1.7.9	HTTP-клиент для отправки запросов к серверу.
cra-template	1.2.0	Шаблон по умолчанию для create-react-app, содержит базовую структуру и конфигурации.
react	19.1.0	Основная библиотека React для построения пользовательского интерфейса.
react-dom	19.1.0	Работа с DOM для приложений на React. Связывает React-компоненты с HTML-страницей.
react-router-dom	7.1.1	Библиотека маршрутизации (роутинга) в React. Позволяет переключать страницы без перезагрузки.
jwt-decode	4.0.0	Библиотека для декодирования JWT-токенов.
react-scripts	3.0.1	Набор скриптов и конфигураций, используемых в create-react-app (сборка, запуск, тесты и т. д.).
socket.io-client	4.8.1	Клиентская часть socket.io, используется для подключения к серверу WebSocket.
web-vitals	4.2.4	Метрики производительности сайта (LCP, FID, CLS и др.). Используется для мониторинга UX.

Окончание таблицы 3.6

webrtc-adapter	9.0.3	Библиотека для кросс-браузерной совместимости WebRTC (используется в видеозвонках, стриминге и т. д.).
----------------	-------	--

Программные библиотеки позволяют упростить реализацию web-приложения. Например, jwt-decode [9] предоставляет готовые легко настраиваемые методы для декодирования JWT-токенов или axios [6] — HTTP-клиент на основе Promise для Node.js, позволяющий легко описывать HTTP-запросы.

### 3.5 Структура серверной части

Основные компоненты структуры серверной части в NestJS основаны на модульной архитектуре, которая обеспечивает гибкость и масштабируемость приложения. В основе NestJS лежат модули, которые группируют связанные функциональные элементы, такие как:

- контроллеры — отвечают за обработку HTTP-запросов и маршрутизацию, определяя, какие методы будут вызваны для обработки конкретных маршрутов. [7]
- сервисы — используются для реализации бизнес-логики и предоставления данных контроллерам. [8].
- репозитории — используются для прямого обращения к базе данных и предоставления данных сервисам. [9]
- guards — промежуточные обработчики, используемые для обеспечения безопасности. [10]

В таблице 3.7 приведён список директорий проекта разработки серверной части web-приложения и назначение файлов, хранящихся в этих директориях.

Таблица 3.7 – Директории серверной части web-приложения

Директория	Назначение
config	Содержит константы конфигурации приложения.
controllers	Содержит контроллеры, которые обрабатывают HTTP-запросы, выполняют бизнес-логику через сервисы и возвращают ответы клиенту.
dto	Содержит файлы, описывающие Data Transfer Object (DTO).
shared	Содержит декораторы, перечисления, гарды, интерфейсы, типы и инструменты для различных действий в разных местах серверной части.
strategies	Содержит стратегии для реализации гибкой, расширяемой и модульной системы аутентификации и авторизации.
guards	Содержит промежуточное ПО для авторизации и проверки токенов.
modules	Содержит модули приложения.

Окончание таблицы 3.7

repositories	Содержит репозитории.
entities	Содержит определения сущностей базы данных.
services	Содержит сервисы.

Таблица соответствия маршрутов к контроллерам и функциям в исходном коде представлена в таблице 3.8.

Таблица 3.8 – Контроллеры и их маршруты

Метод	Маршрут	Контроллер	Метод Контроллера
POST	/auth/register	Auth	register
POST	/auth/login	Auth	login
POST	/auth/profile	Auth	getProfile
POST	/users	Users	create
GET	/users/	Users	getUsers
GET	/users/count	Users	getUserCount
GET	/users/export	Users	exportUsers
GET	/users/most-active	Users	getMostActiveUsers
GET	/users/:id	Users	findOne
DELETE	/users/:id	Users	remove
PATCH	/users/:id	Users	updateUser
POST	/users/:import	Users	importUsers
POST	/channels	Channels	create
PUT	/channels/:id	Channels	update
GET	/channels	Channels	findAll
GET	/channels/id/:id	Channels	findOne
GET	/channels/tag/:tag	Channels	findByTag
DELETE	/channels/:id	Channels	Remove
POST	/messages	Messages	create
GET	/messages	Messages	findConversationMessages
GET	/messages/all	Messages	findAll
PATCH	/messages/:id	Messages	update
DELETE	/messages/:id	Messages	remove
POST	/contacts	Contacts	create
GET	/contacts	Contacts	findAll
GET	/contacts/:id	Contacts	findOne
DELETE	/contacts	Contacts	remove

При передаче данных между клиентом и сервером используется формат. Код контроллера User представлен в приложении Б.

### 3.6 Реализация функций для абонента

#### 3.6.1 Регистрация



Для абонента доступна регистрация, которая позволяет ему создать учетную запись в системе. Этот процесс реализован в методе `register` контроллера `AuthController`. Реализация метода представлена на листинге 3.5.

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}
  @Post('register')
  async register(@Body() registerDto: RegisterDto) {
    return this.authService.register(registerDto);
  }
  ...
}
```

Листинг 3.5 – Реализация метода `register`

Метод принимает HTTP-запрос с данными типа `RegisterDto`. Вызывает метод `register` сервиса `AuthService`, который записывает данные в базу данных, генерирует токены и возвращает их. В результате выполнения, клиенту возвращается объект содержащий созданного пользователя и токены для аутентификации.

### 3.6.2 Аутентификация

Абонент может аутентифицироваться в системе, введя свой email и пароль. Этот процесс реализован в методе `login` контроллера `AuthController`. Реализация метода представлена на листинге 3.6:

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}
  @Post('login')
  @UseGuards(LocalAuthGuard)
  async login(@Request() req) {
    return this.authService.login(req.user);
  }
  ...}
}
```

Листинг 3.6 – Реализация метода `login`

Метод принимает HTTP-запрос с данными. Вызывает метод `login` сервиса `AuthService`, который проверяет существование пользователя и правильность предоставленных данных, а также генерирует токены и возвращает их. В результате выполнения, клиенту возвращается объект, содержащий пользователя и токены для аутентификации.

### 3.6.3 Добавление пользователя в контакт

После успешной аутентификации в системе, абонент может добавить пользователя в контакт, зная его имя. Этот процесс реализован в методе `create` контроллера `ContactController`. Реализация метода представлена на листинге 3.7:

```
@UseGuards(JwtAuthGuard)
@Controller('contacts')
export class ContactController {
  constructor(private readonly contactService: ContactService) {}
  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.USER)
  @Post()
  create(
    @Body() createContactDto: CreateContactDto,
    @Req() req: RequestWithUser,
  ) {
    return this.contactService.create(createContactDto, req.user.id);
  }
}
```

Листинг 3.7 – Реализация метода `create`

Метод принимает HTTP-запрос с данными по `CreateContactDto`. Вызывает метод `create` сервиса `ContactService`, который проверяет полученные данные и в случае успеха добавляет пользователя в контакт.

### 3.6.4 Просмотр списка контактов

После добавления пользователя в контакты, абонент может просмотреть все существующие контакты с помощью метода `findAll` в `ContactController`. Реализация метода представлена на листинге 3.8:

```
@UseGuards(JwtAuthGuard)
@Controller('contacts')
export class ContactController {
  constructor(private readonly contactService: ContactService) {}
  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.USER)
  @Get()
  async findAll(@Req() req: RequestWithUser) {
    return this.contactService.findAll(req.user.id);
  }
}
```

Листинг 3.8 – Реализация метода `findAll`

Метод принимает HTTP-запрос, вызывает метод `findAll` сервиса `ContactService` и выводит все существующие контакты.

### 3.6.5 Удаление контакта

В случае, если абонент захочет удалить существующий контакт, он может воспользоваться методом `remove` в `ContactController`. Реализация метода представлена на листинге 3.9:

```
@Controller('contacts')
export class ContactController {
  constructor(private readonly contactService: ContactService) {}
  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.USER) @Delete()
  remove(
    @Body() removeContactDto: CreateContactDto,
    @Request() req: RequestWithUser,
  ) {
    return this.contactService.remove(removeContactDto.userId,
    req.user.id);}}}
```

Листинг 3.9 – Реализация метода `remove`

Метод принимает HTTP-запрос, вызывает метод `remove` сервиса `ContactService`, удаляет существующий контакт и возвращает этот объект.

### 3.6.6 Отправление и получение сообщения

Абонент может отправить сообщение любому существующему контакту, с помощью метода `create` контроллера `MessageController`. Также абонент может просмотреть имеющиеся сообщения с определенным контактом, с помощью метода `findConversationMessages` в том же контроллере. Реализация обоих методов представлена на листинге 3.10:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER)
@Post()
@UseInterceptors(FileInterceptor('file'))
create(
  @UploadedFile() file: Express.Multer.File,
  @Body() createMessageDto: CreateMessageDto,
  @Request() req: RequestWithUser,
) {
  if (
    !createMessageDto.receiverUserId &&
    !createMessageDto.receiverChannelId
  ) {
    console.error(
      'Missing receiverUserId or receiverChannelId in DTO:',
      createMessageDto,
    );
  }
  return this.messageService.create(createMessageDto, req.user.id,
  file);
}
```

```

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER)
@Get()
findConversationMessages(
  @Request() req: RequestWithUser,
  @Query('receiverUserId') receiverUserId?: string,
  @Query('receiverChannelId') receiverChannelId?: string,
) {
  const senderId = req.user.id;
  if (receiverUserId && receiverChannelId) {
    throw new BadRequestException(
      'Provide either receiverUserId or receiverChannelId, not both.',
    );
  }
  ...
}

```

Листинг 3.10 – Реализация методов create и findConversationMessages

Методы принимают HTTP-запросы, вызывающие методы create и findConversationMessages или findChannelMessages (в зависимости от выбора чата) сервиса MessageService, которые создают сообщения и возвращают их.

### 3.6.7 Редактирование и удаление сообщений

Абонент может редактировать или удалять сообщения, при помощи методов update и remove контроллера MessageController. Реализация методов представлена на листинге 3.11:

```

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER)
@Patch('/:id')
update(@Param('id') id: string, @Body() updateMessageDto:
UpdateMessageDto) {
  return this.messageService.update(+id, updateMessageDto);
}
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER)
@Delete('/:id')
remove(@Param('id') id: string) {
  return this.messageService.remove(+id);
}

```

Листинг 3.11 – Реализация методов update и remove

Методы принимают HTTP-запросы с необходимым id сообщения, если сообщение существует, то применяются методы update и remove сервиса MessageService, которые редактируют или удаляют выбранное сообщение.

### 3.6.8 Просмотр списка каналов

Абонент может просмотреть список каналов, при помощи метода `findAll` контроллера `ChannelController`. Реализация метода представлена на листинге 3.12:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER, UserRole.ADMIN) @Get()
async findAll(@Query() query: { page?: number; limit?: number }) {
  const { page = 1, limit = 10 } = query;
  return await this.channelService.findAllWithPagination(page,
limit);
}
```

Листинг 3.12 – Реализация метода `findAll`

Метод принимает HTTP-запрос, который выводит список каналов.

### 3.6.9 Просмотр и изменение коллекции настроек

Абонент может изменять цвет фона и язык интерфейса в web-приложении, с помощью метода `updateUser`. Реализация метода представлена в листинге 3.13:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER, UserRole.ADMIN, UserRole.MAIN_ADMIN)
@Patch('/:id')
async updateUser(
  @Param('id') id: string,
  @Body() dto: UpdateUserDto,
  @Request() req: { user: { id: number; role: UserRole } },
) {
  const userId = +id;
  const { id: currentUserId, role } = req.user;
  if (role === UserRole.USER) {
    if (currentUserId !== userId) {
      throw new ForbiddenException('You can only update your own
profile');
    }
    const { theme, language, ...rest } = dto;
    if (Object.keys(rest).length > 0) {
      throw new ForbiddenException('You can only update theme and
language');
    }
    return this.userService.updateUserSettings(
      userId,
      { theme, language },
      role,
    );
  }
}
```

Листинг 3.13 – Реализация метода `updateUser`

Метод принимает HTTP-запрос с id текущего абонента, если абонент существует, вызывается метод updateUserSettings сервиса UserService и в зависимости от выбранной настройки, измениться либо цвет фона интерфейса, либо язык интерфейса.

## 3.7 Реализация функций для администратора

### 3.7.1 Аутентификация

Аутентификация администратора осуществляется с помощью точно такого же метода login в контроллере AuthController, как и у абонента. Смотреть листинг 3.6.

### 3.7.2 Просмотр списка пользователей

Администратор может просмотреть список всех пользователей с помощью метода getUsers контроллера UserController. Реализация метода представлена на листинге 3.14:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN, UserRole.ADMIN, UserRole.USER)
@Get()
async getUsers(
  @Query() query: GetUsersQueryDto,
  @Request() req: Request & { user: JwtPayload },
) {
  const currentUser = req.user;
  const { page = 1, limit = 10 } = query;
  try {
    if (currentUser.role === UserRole.ADMIN) {
      return await
this.userService.findAllUsersByRoleWithPagination(
UserRole.USER, page, limit,
);
    }
    return await this.userService.getUsersWithFilters(query);
  } catch (error) {
    console.error('Error fetching users:', error);
    throw new HttpException(
      'Error fetching users',
      HttpStatus.INTERNAL_SERVER_ERROR,
    );
  }
}
```

Листинг 3.14 – Реализация метода update

Метод принимает HTTP-запрос. Вызывает метод findAllUsersByRoleWithPagination сервиса UserService, который выводит все пользователей с пагинацией.

### 3.7.3 Блокировать и разблокировать пользователей

Администратор может заблокировать и разблокировать абонента в web-приложении, с помощью метода `updateUser` контроллера `UserController`. Реализация метода представлена на листинге 3.15:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER, UserRole.ADMIN, UserRole.MAIN_ADMIN)
@Patch('/:id')
async updateUser(
  @Param('id') id: string,
  @Body() dto: UpdateUserDto,
  @Request() req: { user: { id: number; role: UserRole } },
) {
  const userId = +id;
  const { id: currentUserId, role } = req.user;
  if (role === UserRole.ADMIN) {
    const { status, ...rest } = dto;
    if (Object.keys(rest).length > 0 || !status) {
      throw new ForbiddenException('Admins can only update user status');
    }
    return this.userService.updateUserStatus(userId, status, role);
  }
}
```

Листинг 3.15 – Реализация метода `updateUser`

Метод принимает HTTP-запрос с `id` абонента и вызывает метод `updateUserStatus` сервиса `UserService`, который изменяет статус пользователя с `active` на `blocked` при блокировке и наоборот, при разблокировке.

### 3.7.4 Добавление, удаление и редактирование канала

Администратор может добавить новый канал, удалить и редактировать существующий канал, с помощью методов `create`, `update` и `remove` контроллера `ChannelController`. Реализация методов представлена в листинге 3.16:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.ADMIN) @Post()
create(@Body() dto: CreateChannelDto) {
  return this.channelService.create(dto); }
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.ADMIN) @Patch('/:id')
update(@Param('id') id: string, @Body() dto: UpdateChannelDto) {
  return this.channelService.update(+id, dto);
}
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.ADMIN) @Delete('/:id')
```

```

remove(@Param('id') id: string) {
    return this.channelService.remove(+id)
}

```

Листинг 3.16 – Реализация методов create, update и remove

Методы принимают HTTP-запросы (с GET принимают запросы с id существующего канала) и вызывают методы create, update и remove сервиса ChannelService, которые создают канал, редактируют его и удаляют.

## 3.8 Реализация функций для главного администратора

### 3.8.1 Аутентификация

Аутентификация главного администратора в системе осуществляется также как и абонента, с помощью метода login в AuthController. Реализацию метода смотреть в листинге 3.6.

### 3.8.2 Просмотр списка всех абонентов и администраторов

Главный администратор может просмотреть список всех абонентов и администраторов, с помощью метода getUsers контроллера UserController. Реализация метода представлена на листинге 3.17:

```

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN, UserRole.ADMIN, UserRole.USER)
@Get()
async getUsers(
    @Query() query: GetUsersQueryDto,
    @Request() req: Request & { user: JwtPayload },
) {
    const currentUser = req.user;
    const { page = 1, limit = 10 } = query;
    try {
        if (currentUser.role === UserRole.MAIN_ADMIN) {
            return await
this.userService.findAllUsersWithPagination(page, limit);
        }
        return await this.userService.getUsersWithFilters(query);
    } catch (error) {
        console.error('Error fetching users:', error);
        throw new HttpException(
            'Error fetching users',
            HttpStatus.INTERNAL_SERVER_ERROR,
        );
    }
}

```



### Листинг 3.17 – Реализация метода getUsers

Метод принимает HTTP-запрос. Вызывает метод `findAllUsersWithPagination` сервиса `UserService`, который выводит список всех абонентов и администраторов.

### 3.8.3 Удаление абонентов

Главный администратор может удалить абонента из системы web-приложения, с помощью метода `remove` контроллера `UserController`. Реализация метода представлена на листинге 3.18

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Delete('/:id')
async remove(@Param('id') id: string) {
  return await this.userService.removeUser(+id);
}
```

### Листинг 3.18 – Реализация метода remove

Метод принимает HTTP-запрос с `id` абонента, который вызывает метод `removeUser` сервиса `UserService`, который удаляет абонента из системы.

### 3.8.4 Изменение ролей пользователей

Главный администратор может наделить абоненту права администратора и наоборот: снять с администратора права, за счет чего он станет обычным абонентом, с помощью метода `updateUser` контроллера `UserController`. Реализация метода представлена на листинге 3.19:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.USER, UserRole.ADMIN, UserRole.MAIN_ADMIN)
@Patch('/:id')
async updateUser(
  @Param('id') id: string,
  @Body() dto: UpdateUserDto,
  @Request() req: { user: { id: number; role: UserRole } },
) {
  const userId = +id;
  const { id: currentUserId, role } = req.user;
  if (role === UserRole.MAIN_ADMIN) {
    const { role: newRole, ...rest } = dto;
    if (Object.keys(rest).length > 0 || !newRole) {
      throw new ForbiddenException('Main admins can only update user role');
    }
    return this.userService.updateUserRole(userId, newRole, role);
  }
}
```

### Листинг 3.19 – Реализация метода updateUser

Метод принимает HTTP-запрос с id абонента или же администратора, вызывает метод updateUserRole сервиса UserService, который изменяет роль выбранного пользователя.

### 3.8.5 Просмотр наиболее активных пользователей, общего количества пользователей

Главный администратор может просмотреть, сколько на данный момент в web-приложении существует абонентов, а также может посмотреть, какие абоненты самые активные (по количеству сообщений), с помощью методов getUserCount и getMostActiveUsers контроллера UserController. Реализация методов представлена на листинге 3.20:

```
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Get('count')
async getUserCount() {
  const count = await this.userService.countAllUsers();
  return { count };
}
@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Get('most-active')
async getMostActiveUsers(
  @Request() req: any,
  @Query('page') page: number = 1,
  @Query('limit') limit: number = 10,
) {
  return await this.userService.findMostActiveUsers(page, limit);
}
```

### Листинг 3.20 – Реализация методов getUserCount и getMostActiveUsers

Оба метода принимают HTTP-запросы и вызывают методы countAllUsers и findMostActiveUsers сервиса UserService, которые возвращают количество абонентов в web-приложении и выводят список наиболее активных пользователей.

## 3.9 Структура клиентской части

Клиентская часть приложения реализована с использованием компонентного подхода. Основная логика и элементы пользовательского интерфейса размещены в директории src. Директории представлены в таблице 3.8.

Таблица 3.8 – Директории клиентской части web-приложения

Директория	Назначение
------------	------------

## Окончание таблицы 3.8

components	Содержит переиспользуемые React-компоненты интерфейса, такие как формы, кнопки, карточки, модальные окна и т.д.
utils	Содержит вспомогательные функции и утилиты, используемые в разных частях приложения. Например, auth.js реализует функции авторизации, работы с токенами, проверки сессий и т.д.

Таблица соответствия маршрутов и компонентов страниц представлена в таблице 3.9.

Таблица 3.9 – Маршруты и компоненты страниц

Компонент страницы	Маршрут	Роль	Назначение компонента
Login & Register	/auth/login /auth/register	Абонент, администратор, главный администратор	Страницы, содержащие формы регистрации и аутентификации.
ChannelsList	/channels	Абонент	Страница со списком каналов.
ContactsList	/contacts	Абонент	Страница со списком контактов.
ChatWindow	/messages	Абонент	Страница для отправки и получения сообщений, видеозвонка.
ProfileSettings	/users/:userId	Абонент	Страница для изменения коллекции настроек, а также выхода из аккаунта.
AdminDashboard	/admin	Администратор	Страница для модерации администратора.
MainAdminDashBoard	/main-admin	Главный администратор	Страница для полного администрирования.

Листинг кода компонента App представлен в приложении В.

Листинг кода компонента Login представлен в приложении Г.

### 3.11 Выводы по разделу

Веб-приложение разработано на современном технологическом стеке, обеспечивающем высокую производительность, надежность и масштабируемость. В качестве основы серверной части была выбрана популярная программная платформа Node.js, которая благодаря своей событийно-ориентированной архитектуре и асинхронной модели ввода-вывода идеально подходит для создания высоконагруженных сетевых приложений. Для структурирования серверного кода использовался прогрессивный фреймворк NestJS. NestJS сочетает в себе элементы объектно-ориентированного программирования, функционального программирования и реактивного программирования, что значительно ускоряет процесс разработки и упрощает поддержку кода.

Для хранения данных была выбрана реляционная база данных Postgres, известная своей надежностью, соответствием стандартам SQL и расширенными возможностями. Postgres обеспечивает полную поддержку ACID-транзакций, сложные типы данных, мощные механизмы индексирования и богатый набор функций для работы с JSON. Взаимодействие с базой данных осуществляется через TypeORM - продвинутый ORM-инструмент, который позволяет работать с базой данных используя объектно-ориентированный подход. TypeORM поддерживает автоматическую миграцию схемы базы данных, кеширование запросов, ленивую загрузку связей и множество других полезных функций, что значительно упрощает работу с данными.

Клиентская часть приложения разработана с использованием библиотеки React - одного из самых популярных решений для создания пользовательских интерфейсов. React реализует компонентный подход, позволяющий разбивать интерфейс на независимые, переиспользуемые компоненты, каждый из которых управляет своим состоянием. Это не только способствует поддержанию порядка в коде, но и значительно упрощает процесс разработки сложных интерфейсов.

В приложении был тщательно проработан функционал для трех ролей пользователей: абонента, администратора и главного администратора. Для каждой роли реализован уникальный набор из более чем 20 функций, охватывающих все аспекты работы мессенджера.

## 4 Тестирования web-приложения

### 4.1 Функциональное тестирование

Для проверки корректности работы всех функций разработанного web-приложения было проведено ручное тестирование, описание и итоги которого представлены в таблице 4.1.

Таблица 4.1 – Описание тестирования функций web-приложения

Номер	Функция web-приложения	Описание тестирования	Ожидаемый результат	Итог тестирования функции
1	Регистрация	Действие: отправить POST запрос на адрес /auth/register, указав в теле запроса email со значением «usercontact1@example.com», имя пользователя со значением «USERcontact1», пароль со значением «USERcontact1».	Сервер должен вернуть ответ в формате JSON с созданным пользователем и access-токеном.	Успешно
2	Аутентификация	Действие: отправить POST запрос на адрес /auth/login, указав в теле запроса email со значением «usercontact1@example.com» и пароль со значением «USERcontact1».	Сервер должен вернуть ответ в формате JSON с данными пользователя и access-токеном.	Успешно
3	Добавление пользователя в контакт	Действие: отправить POST запрос на адрес /contacts, указав в теле запроса идентификатор пользователя «contactUserId: 16».	Сервер должен вернуть созданный контакт со статусом 200.	Успешно
4	Удаление контакта	Действие: отправить DELETE запрос на адрес /contacts/:id, указав в id значение «1».	Сервер должен вернуть JSON с удаленным объектом.	Успешно

Продолжение таблицы 4.1

5	Добавление сообщения	Действие: отправить POST запрос на адрес /messages, в теле запроса указать receiverUserId или receiverChannelId со значением «15» и «1» соответственно, а также указать text со значением «hello».	Сервер должен вернуть ответ со статус кодом 200 и JSON с объектом созданного сообщения.	Успешно
6	Редактирование сообщения	Действие: отправить PATCH запрос на адрес /messages/:id, где id со значением «1», а в теле запроса: text со значением «ababababa».	Сервер должен вернуть ответ в формате JSON с объектом измененного сообщения.	Успешно
7	Удаление сообщения	Действие: отправить DELETE запрос по адресу /messages/:id, где id со значением «1».	Сервер должен вернуть ответ в формате JSON с удаленным объектом сообщения.	Успешно
8	Просмотр сообщений в чате пользователя или канала	Действие: отправить GET запрос на адрес /messages с query параметром receiverUserId или receiverChannelId со значением «15» и «1» соответственно.	Сервер должен вернуть ответ в формате JSON с массивом всех объектов сообщений пользователя.	Успешно
9	Просмотр списка каналов	Действие: отправить GET запрос на адрес /channels. Ожидаемый результат:	Сервер должен вернуть ответ в формате JSON с массивом всех объектов каналов.	Успешно
10	Просмотр списка контактов	Действие: отправить GET запрос на адрес /contacts. Ожидаемый результат:	Сервер должен вернуть ответ в формате JSON с массивом всех объектов контактов пользователя.	Успешно

Продолжение таблицы 4.1

11	Изменение коллекции настроек	Действие: отправить PATCH запрос на адрес /users/:id указав в id значение «16», а также в теле запроса указать theme и language со значениями «dark» и «ru» соответственно.	Сервер должен вернуть ответ в формате JSON с измененным объектом коллекции настроек.	Успешно
12	Создание каналов	Действие: отправить POST запрос на адрес /channels, в теле запроса указать name и tag со значениями «firstchannel» и «first_channel» соответственно.	Сервер должен вернуть ответ в формате JSON с созданным объектом канала.	Успешно
13	Редактировать канал	Действие: отправить PATCH запрос по адресу /channels/:id, указать id со значением «1», в теле указать объект с полем tag и значением «new_first_channel».	Сервер должен вернуть ответ со статусом 201 в формате JSON с измененным объектом канала.	Успешно
14	Удалить канал	Действие: отправить DELETE запрос по адресу /channels/:id, указав id со значением «1».	Сервер должен вернуть ответ со статусом 200 в формате JSON с удаленным объектом канала.	Успешно
15	Заблокировать пользователя	Действие: отправить PATCH запрос по адресу /users/:id, указав id со значением 16, в теле передать status со значением «blocked».	Сервер должен вернуть ответ со статусом 200 в формате JSON и измененным объектом пользователя.	Успешно
16	Разблокировать пользователя	Действие: отправить PATCH запрос по адресу /users/:id, указав id со значением 16, в теле передать status со значением «active».	Сервер должен вернуть ответ со статусом 200 в формате JSON и измененным объектом пользователя.	Успешно

## Окончание таблицы 4.1

17	Просмотр списка всех пользователей	Действие: отправить GET запрос по адресу /users.	Сервер должен вернуть ответ со статусом 200 в формате JSON и массив объектов всех пользователей с ролью «user».	Успешно
18	Просмотр списка всех пользователей и администраторов	Действие: отправить GET запрос по адресу /users.	Сервер должен вернуть ответ со статусом 200 в формате JSON с массивом объектов всех пользователей с ролью «user» и «admin».	Успешно
19	Удаление пользователя	Действие: отправить DELETE запрос по адресу /users/:id, указав id со значением «3».	Сервер должен вернуть ответ со статусом 200 в формате JSON с удаленным объектом пользователя.	Успешно
20	Изменение роли пользователя	Действие: отправить PATCH запрос по адресу /users/:id, указав id со значением «4», в теле передать role со значением «admin».	Сервер должен вернуть ответ в формате JSON с измененным объектом пользователя.	Успешно

Таким образом, были протестированы все ключевые функции web-приложения.

## 4.2 Выводы по разделу

1. Проведено ручное тестирование всех ключевых функций web-приложения. Корректность работы системы подтверждена соответствием фактических результатов тестирования ожидаемым. Количество тестов составило 20.



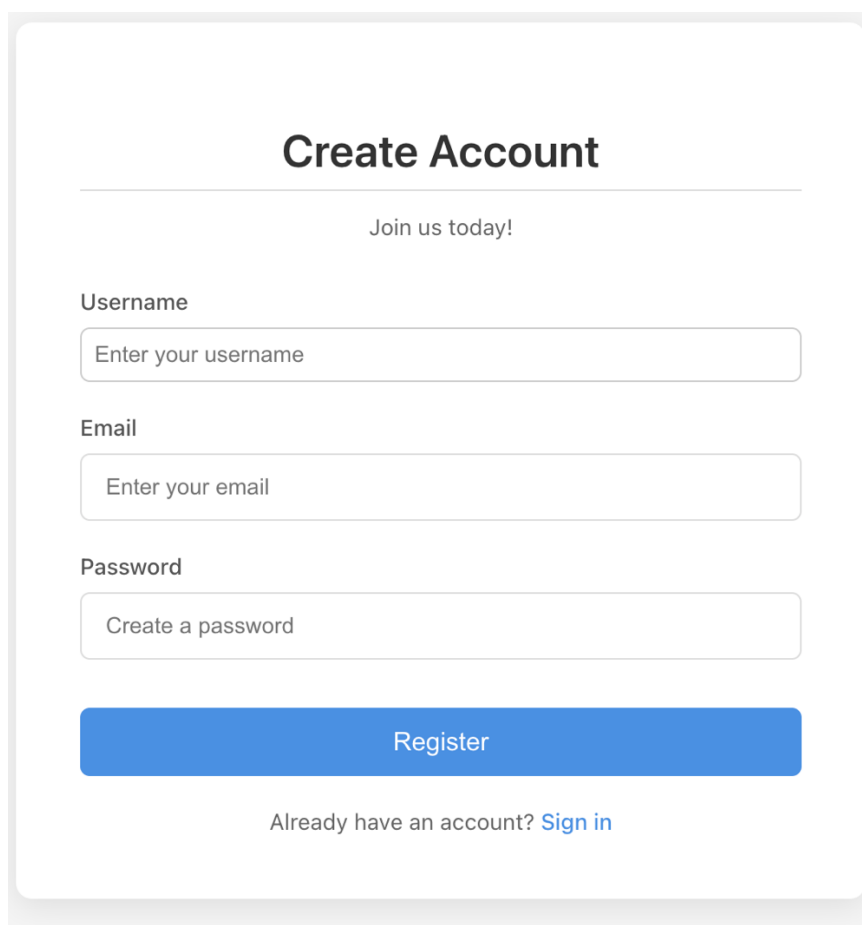
## 5 Руководство пользователя

### 5.1 Руководство абонента

#### 5.1.1 Регистрация

При открытии сайта абонент автоматически попадает на форму аутентификации и он может перейти на форму регистрации, нажав на кнопку «Sign up».

Форма регистрации представлена на рисунке 5.1.



The image shows a registration form titled "Create Account". Below the title is a horizontal line and the text "Join us today!". There are three input fields: "Username" with the placeholder "Enter your username", "Email" with the placeholder "Enter your email", and "Password" with the placeholder "Create a password". Below these fields is a blue button labeled "Register". At the bottom, there is a link that says "Already have an account? [Sign in](#)".

Рисунок 5.1 – Форма регистрации

Здесь необходимо заполнить форму, указав свое имя, электронную почту и пароль.

#### 5.1.1 Аутентификация

При открытии сайта гость автоматически попадает на форму аутентификации. Форма аутентификации представлена на рисунке 5.2.

The image shows a login interface with a light gray background. At the top, the text 'Welcome Back' is centered in a bold, black font. Below it, a horizontal line separates the header from the login instructions. The text 'Sign in to continue' is centered below the line. There are two input fields: the first is labeled 'Email' and contains the placeholder text 'Enter your email'; the second is labeled 'Password' and contains the placeholder text 'Enter your password'. Below these fields is a large, solid blue button with the text 'Sign In' in white. At the bottom, the text 'Don't have an account?' is followed by a blue link labeled 'Sign up'.

Рисунок 5.2 – Форма аутентификации

Здесь необходимо заполнить форму, указав свою электронную почту и пароль. При нажатии на кнопку «Sign In» будет произведена попытка аутентификации.

После успешной аутентификации, абонент переходит на главную страницу, состоящую из нескольких компонентов, а администратор и главный администратор переходят на страницы /admin и /main-admin.

## 5.2 Руководство администратора

### 5.2.1 Добавление канала

Для добавления канала администратор должен пройти аутентификацию, после чего в открывшемся окне будет виден весь его функционал. Страница администратора представлена на рисунке 5.3.

## Admin Dashboard

### Users Management

Name	Email	Status	Actions
Evgeniy	eugen@gmail.com	active	<a href="#">Block</a>
a	a@a.com	active	<a href="#">Block</a>
USER	user@example.com	active	<a href="#">Block</a>
NEW_USER	newuser@gmail.com	active	<a href="#">Block</a>
USERcontact2	usercontact2@example.com	active	<a href="#">Block</a>
<a href="#">Previous</a> Page 1 of 3 <a href="#">Next</a>			

### Channel Management

#### Add New Channel

<input type="text" value="Hobbies"/>	<input type="text" value="hobby"/>	<input type="button" value="Add Channel"/>		
ID	Name	Tag	Actions	
1	firstchannel	new_first_channel	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
5	secondchannel	new_second_channel	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
8	ccc	ccc	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
7	aaa	aaa	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>
<input type="button" value="Previous"/> Page 1 of 1 <input type="button" value="Next"/>				

Рисунок 5.3 – Страница профиля администратора

Каналы у администратора будут отображаться в нижней части страницы, чтобы его создать, нужно нажать на кнопку «Add Channel». После этого созданный канал отобразится в списке каналов на рисунке 5.4.

## Channel Management

### Add New Channel

Channel Name		Channel Tag	Add Channel	
ID	Name	Tag	Actions	
1	firstchannel	new_first_channel	Edit	Delete
5	secondchannel	new_second_channel	Edit	Delete
8	ccc	ccc	Edit	Delete
7	aaa	aaa	Edit	Delete
10	Hobbies	hobby	Edit	Delete
<div>PreviousPage 1 of 1Next</div>				

Рисунок 5.4 – Страница создания канала

Администратор выполняет все свои функции на одной странице.

### 5.2.2 Изменение канала

Канал отображается в списке каналов администратора, чтобы изменить название или тег канала необходимо нажать на кнопку «Edit» и в появившихся полях выполнить необходимые изменения, после чего нажать на кнопку «Save» для сохранения или «Cancel» для отмены изменений изменение канала представлено на рисунке 5.5

ID	Name	Tag	Actions	
1	firstchannel	new_first_channel	Edit	Delete
5	secondchannel	new_second_channel	Edit	Delete
8	ccc	ccc	Edit	Delete
7	<input type="text" value="Movies"/>	<input type="text" value="movie"/>	Save	Cancel
10	Hobbies	hobby	Edit	Delete
<input type="button" value="Previous"/> Page 1 of 1 <input type="button" value="Next"/>				

Рисунок 5.5 – Отображение изменений в канале

Для изменения существующего канала нужно нажать на кнопку «Save», после чего измененный канал автоматически отобразится в списке каналов администратора. Внешний вид списка каналов с измененным каналом представлен на рисунке 5.6.

ID	Name	Tag	Actions	
1	firstchannel	new_first_channel	Edit	Delete
5	secondchannel	new_second_channel	Edit	Delete
8	ccc	ccc	Edit	Delete
10	Hobbies	hobby	Edit	Delete
7	Movies	movie	Edit	Delete
<input type="button" value="Previous"/> Page 1 of 1 <input type="button" value="Next"/>				

Рисунок 5.6 – Список каналов с изменениями

Таким образом администратор может изменить данные о том или ином канале.

### 5.2.3 Удаление канала

Для удаления канала администратору нужно нажать кнопку «Delete» в области отображении информации о канале, рядом с кнопкой «Edit» и в высветившемся окне нажать на кнопку «ОК». Отображение окна с подтверждением действия представлено на рисунке 5.7.

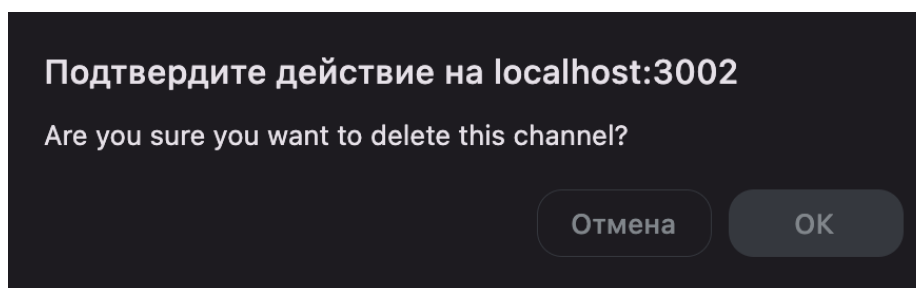


Рисунок 5.7 – Окно с подтверждением удаления канала

После удаления канала, список каналов обновится и удаленный канал пропадет.

### 5.2.4 Блокирование пользователя

Администратор может изменить статус пользователя, тем самым заблокировав ему доступ ко всему web-приложению, для этого необходимо в списке пользователей нажать на кнопку «Block». Список пользователей с кнопкой «Block» представлен на рисунке 5.8.

Name	Email	Status	Actions
Evgeniy	eugen@gmail.com	active	<button>Block</button>
a	a@a.com	active	<button>Block</button>
USER	user@example.com	active	<button>Block</button>
NEW_USER	newuser@gmail.com	active	<button>Block</button>
USERcontact2	usercontact2@example.com	active	<button>Block</button>
<div> <span>Previous</span> <span>Page 1 of 3</span> <span>Next</span> </div>			

Рисунок 5.8 – Список пользователей с кнопкой «Block»

После нажатия на кнопку блокировки, статус пользователя измениться на значение «blocked». Изменения представлены на рисунке 5.9

Name	Email	Status	Actions
Evgeniy	eugen@gmail.com	active	<button>Block</button>
a	a@a.com	blocked	<button>Unblock</button>
USER	user@example.com	active	<button>Block</button>
NEW_USER	newuser@gmail.com	active	<button>Block</button>
USERcontact2	usercontact2@example.com	active	<button>Block</button>
<div> <span>Previous</span> <span>Page 1 of 3</span> <span>Next</span> </div>			

Рисунок 5.9 – Список пользователей с заблокированным пользователем

Таким образом администратор может блокировать пользователей за различные нарушения. Чтобы разблокировать пользователя в системе, необходимо нажать на кнопку «UnBlock» рядом со статусом пользователя. После этого пользователь сможет дальше активно пользоваться web-приложением.

### 5.3 Руководство главного администратора

#### 5.3.1 Просмотр пользователей и администраторов, изменение роли пользователя

Главный администратор может просмотреть всех пользователей и администраторов в приложении после успешной аутентификации в приложении. Страница главного администратора представлена на рисунке 5.10

# Main Admin Dashboard

Total Users: 12

Most Active Users

Name	Email	Status	Role	Message Count
USERcontact2	usercontact2@example.com	active	user	55
USERcontact1	usercontact1@example.com	active	user	9
NEW_USER	newuser@gmail.com	active	user	6
TEST_MAIN_ADMIN	testmainadmin@example.com	active	main_admin	0
Evgeniy	eugen@gmail.com	active	user	0
a	a@a.com	blocked	user	0
USER	user@example.com	active	user	0
AlekseyAdmin	alekseyadmin@example.com	active	admin	0
Maria	mary@gmail.com	active	user	0
aleksey	alexeyorlov2004@example.com	active	user	0

All Users

Name	Email	Status	Role	Actions
MAIN_ADMIN	mainadmin@example.com	active	User	Delete
Evgeniy	eugen@gmail.com	active	User	Delete
a	a@a.com	blocked	User	Delete
USER	user@example.com	active	User	Delete
NEW_USER	newuser@gmail.com	active	User	Delete
USERcontact2	usercontact2@example.com	active	User	Delete
AlekseyAdmin	alekseyadmin@example.com	active	Admin	Delete
USERcontact1	usercontact1@example.com	active	User	Delete
Maria	mary@gmail.com	active	User	Delete
aleksey	alexeyorlov2004@example.com	active	User	Delete

Previous Page 1 Next

Рисунок 5.10 – Страница главного администратора

Чтобы изменить роль пользователя в получившемся списке пользователей и администраторов необходимо в столбце «Role» нажать на список и выбрать одну необходимую роль из трех: «User», «Admin» или «Main Admin». Изменение роли пользователя представлено на рисунке 5.11

Name	Email	Status	Role	Actions
firstuser	fu@gmail.com	active	User ▾	Delete
bbbbbb	b@h.com	active	User ▾	Delete
STUDENT	stud@belstu.by	active	User ▾	Delete
Evgeniy	eugen@gmail.com	active	Admin ▾	Delete
<div> <div>Previous</div> <div>Page 2</div> <div>Next</div> </div>				

Рисунок 5.11 – Измененная роль пользователя «Evgeniy»

Таким образом главный администратор может наделять правами администратора обычных абонентов, а также снимать привилегию администратора.

### 5.3.2 Удаление пользователя

Главный администратор может удалить аккаунт пользователя из системы web-приложения, для этого ему необходимо нажать на кнопку «Delete» в столбце «Actions» в списке всех пользователей и администраторов. После нажать на выветившемся окне подтверждения действия кнопку «ОК». Окно подтверждения удаления представлено на рисунке 5.12

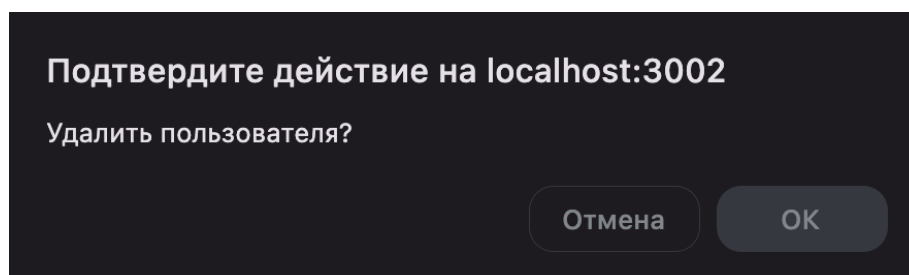


Рисунок 5.19 – Окно с подтверждением удаления пользователя

После подтверждения удаления, выбранный пользователь или администратор будет удален из списка пользователей и администраторов.

## Заключение

В результате работы над проектом было разработано web-приложение «Мессенджер», которое предназначено для свободного, удобного и безопасного обмена сообщениями между пользователями. Web-приложение поддерживает 3 роли: абонент, администратор и главный администратор. Каждая роль имеет уникальный набор прав и возможностей.

Web-приложение включает 13 ключевых функций, охватывающих весь спектр необходимого функционала: регистрация, аутентификация, создание, редактирование и удаление сообщений, просмотр контактов и каналов, изменение настроек пользователя, добавление каналов, удаление контактов и каналов, блокировка и разблокировка абонентов, а также удаление пользователей.

База данных web-приложения представляет собой тщательно спроектированную реляционную структуру, состоящую из четырех взаимосвязанных таблиц, обеспечивающих хранение и обработку всех критически важных данных системы. Основная таблица Users содержит полную информацию о пользователях, включая уникальные идентификаторы, учетные данные (логины и хэшированные пароли), ролевую принадлежность (абонент, администратор, главный администратор), статусы аккаунтов (активный/заблокированный), а также персональные настройки (темы оформления, языковые предпочтения). Таблица Contacts реализует систему связей между пользователями, храня данные о владельцах контактов (`owner_user_id`) и соответствующих контактных пользователях (`contact_user_id`) с временными метками создания связей, обеспечивая при этом целостность данных через внешние ключи. Для организации группового взаимодействия предназначена таблица Channels, которая сохраняет информацию о каналах (уникальные идентификаторы, названия, теги) и обеспечивает быстрый поиск через специализированные индексы. Сердце системы - таблица Messages - содержит всю переписку, включая текстовые сообщения с метаданными (отправители, получатели, временные отметки), файловые вложения (с хранением имен, типов, размеров и URL-путей), а также информацию о доставке и прочтении, при этом для оптимизации запросов созданы составные индексы по наиболее часто используемым полям. Все таблицы связаны между собой через систему внешних ключей с каскадными ограничениями, что гарантирует целостность данных, а применение механизмов транзакций обеспечивает надежность при одновременном доступе множества пользователей.

Архитектура web-приложения имеет несколько ключевых особенностей: серверная часть построена на Node.js с использованием фреймворка NestJS. В качестве клиентской части используется React, что обеспечивает масштабируемость web-приложения.

Общий объем программного кода web-приложения составил 5000 авторских строк. Общее количество тестов составило 20.

В соответствии с полученным результатом работы web-приложения можно сделать вывод, что цель достигнута, а требования технического задания выполнены в полном объеме.



## Список используемых источников

- 1 Полное руководство по Node.js [Электронный ресурс] / Режим доступа: <https://nodejsdev-ru.pages.dev/guides/freecodecamp/> – Дата доступа: 16.09.2024.
- 2 Платформа *teams.live.com* [Электронный ресурс]. – Режим доступа: <https://teams.live.com/> – Дата доступа: 16.09.2024.
- 3 Платформа *web.telegram.org* [Электронный ресурс]. – Режим доступа: <https://web.telegram.org> – Дата доступа: 16.09.2024.
- 4 Руководство *PostgreSQL* [Электронный ресурс]. – Режим доступа: <https://www.postgres.com/resources/products/fundamentals/basics/>. – Дата доступа: 10.10.2024.
- 5 Обзор *NestJS* [Электронный ресурс]. – Режим доступа: <https://docs.nestjs.com/>. – Дата доступа: 15.10.2024.
- 6 Обзор *axios* [Электронный ресурс]. – Режим доступа: <https://www.npmjs.com/package/axios/>. – Дата доступа: 25.10.2024.
- 7 Руководство по контроллерам *NestJS* [Электронный ресурс]. – Режим доступа <https://docs.nestjs.com/controllers>. – Дата доступа: 01.11.2024.
- 8 Руководство по сервисам *NestJS* [Электронный ресурс]. – Режим доступа: <https://docs.nestjs.com/providers/>. – Дата доступа: 01.11.2024.
- 9 Руководство по репозиториям *NestJS* [Электронный ресурс]. – Режим доступа: <https://docs.nestjs.com/repositories/>. – Дата доступа: 02.11.2024.
- 10 Руководство по модулям *NestJS* [Электронный ресурс]. – Режим доступа: <https://docs.nestjs.com/modules/>. – Дата доступа: 04.11.2024.
- 11 Руководство реализации аутентификации с помощью *NestJS* [Электронный ресурс]. – Режим доступа: <https://docs.nestjs.com/security/authentication>. – Дата доступа: 04.11.2024.

## Приложение А

```

import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UserModule } from './user/user.module';
import { MessageModule } from './message/message.module';
import { ContactModule } from './contact/contact.module';
import { ChannelModule } from './channel/channel.module';
import { AuthModule } from './auth/auth.module';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { TypeOrmModule, TypeOrmModuleOptions } from '@nestjs/typeorm';
import configFactory from './config';

@Module({
  imports: [
    UserModule,
    AuthModule,
    ChannelModule,
    ContactModule,
    MessageModule,
    ConfigModule.forRoot({
      envFilePath: '.env',
      isGlobal: true,
      load: [configFactory],
    }),
    TypeOrmModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: (configService: ConfigService):
TypeOrmModuleOptions => {
      const dbConfig =
configService.get<TypeOrmModuleOptions>('dbConfig');
      if (!dbConfig) {
        throw new Error('Database configuration is not defined');
      }
      return dbConfig;
    },
  ]),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

```

Листинг – Код модуля App

## Приложение Б

```

/* eslint-disable @typescript-eslint/no-unsafe-return */
import {
  Controller,
  Post,
  Body,
  UsePipes,
  ValidationPipe,
  Get,
  Param,
  Patch,
  Delete,
  Request,
  UseGuards,
  ForbiddenException,
  Query,
  HttpException,
  HttpStatus,
} from '@nestjs/common';
import { UserService } from '../user.service';
import { CreateUserDto } from '../dto/create-user.dto';
import { UpdateUserDto } from '../dto/update-user.dto';
import { Roles } from 'src/shared/decorators/roles.decorator';
import { JwtAuthGuard } from 'src/auth/guards/jwt-auth.guard';
import { UserRole } from 'src/shared/enums/user-role.enum';
import { GetUsersQueryDto } from '../dto/get-users-query';
import { JwtPayload } from 'src/auth/interfaces/jwt-payload.interface';
import { RolesGuard } from 'src/shared/guards/roles.guard';

@Controller('users')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Post()
  @UsePipes(new ValidationPipe())
  async create(@Body() createUserDto: CreateUserDto) {
    return await this.userService.createUser(createUserDto);
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.MAIN_ADMIN, UserRole.ADMIN, UserRole.USER)
  @Get()
  async getUsers(
    @Query() query: GetUsersQueryDto,
    @Request() req: Request & { user: JwtPayload },
  ) {
    const currentUser = req.user;
    const { page = 1, limit = 10 } = query;

    try {
      if (query.searchTerm) {

```

```

        return await this.userService getUsersWithFilters(query);
    }

    if (currentUser.role === UserRole.MAIN_ADMIN) {
        return await
this.userService.findAllUsersWithPagination(page, limit);
    }

    if (currentUser.role === UserRole.ADMIN) {
        return await
this.userService.findAllUsersByRoleWithPagination(
            UserRole.USER,
            page,
            limit,
        );
    }

    if (currentUser.role === UserRole.USER) {
        return [await this.userService.findOneUser(currentUser.id)];
    }

    return await this.userService getUsersWithFilters(query);
} catch (error) {
    console.error('Error fetching users:', error);
    throw new HttpException(
        'Error fetching users',
        HttpStatus.INTERNAL_SERVER_ERROR,
    );
}
}

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Get('count')
async getUserCount() {
    const count = await this.userService.countAllUsers();
    return { count };
}

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Get('export')
async exportUsers() {
    const users = await this.userService.findAllUsers();
    return { exported: users };
}

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Get('most-active')
async getMostActiveUsers(
    @Request() req: any,
    @Query('page') page: number = 1,

```

```

    @Query('limit') limit: number = 10,
  ) {
    return await this.userService.findMostActiveUsers(page, limit);
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.ADMIN)
  @Get('/:id')
  async findOne(@Param('id') id: string) {
    return await this.userService.findOneUser(+id);
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.MAIN_ADMIN)
  @Delete('/:id')
  async remove(@Param('id') id: string) {
    return await this.userService.removeUser(+id);
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(UserRole.USER, UserRole.ADMIN, UserRole.MAIN_ADMIN)
  @Patch('/:id')
  async updateUser(
    @Param('id') id: string,
    @Body() dto: UpdateUserDto,
    @Request() req: { user: { id: number; role: UserRole } },
  ) {
    const userId = +id;
    const { id: currentUserId, role } = req.user;

    if (role === UserRole.USER) {
      if (currentUserId !== userId) {
        throw new ForbiddenException('You can only update your own profile');
      }

      const { theme, language, ...rest } = dto;
      if (Object.keys(rest).length > 0) {
        throw new ForbiddenException('You can only update theme and language');
      }
      return this.userService.updateUserSettings(
        userId,
        { theme, language },
        role,
      );
    }

    if (role === UserRole.ADMIN) {
      const { status, ...rest } = dto;

      if (Object.keys(rest).length > 0 || !status) {

```

```

        throw new ForbiddenException('Admins can only update user
status');
    }

    return this.userService.updateUserStatus(userId, status,
role);
}

    if (role === UserRole.MAIN_ADMIN) {
        const { role: newRole, ...rest } = dto;
        if (Object.keys(rest).length > 0 || !newRole) {
            throw new ForbiddenException('Main admins can only update
user role');
        }
        return this.userService.updateUserRole(userId, newRole, role);
    }
}

@UseGuards(JwtAuthGuard, RolesGuard)
@Roles(UserRole.MAIN_ADMIN)
@Post('import')
async importUsers(@Body() users: CreateUserDto[]) {
    return await this.userService.bulkCreate(users);
}
}

```

Листинг – Код контроллера UserController

## Приложение В

```

import React, { useState, useEffect, useContext, createContext } from
'react';
import ContactsList from './components/ContactsList';
import ChannelsList from './components/ChannelsList';
import ChatWindow from './components/ChatWindow';
import ProfileSettings from './components/ProfileSettings';
import Login from './components/Login';
import Register from './components/Register';
import ContextMenu from './components/ContextMenu';
import './App.css';
import { getToken, getUserIdFromToken, isTokenExpired, removeToken }
from './utils/auth';

// Создаем контекст для темы и языка
const SettingsContext = createContext();

export const useSettings = () => useContext(SettingsContext);

function App() {
  const [activeTab, setActiveTab] = useState('contacts');
  const [activeChat, setActiveChat] = useState(null);
  const [isProfileOpen, setProfileOpen] = useState(false);
  const [isLoggedIn, setIsLoggedIn] = useState(!!getToken() &&
!isTokenExpired());
  const [isAddContactModalOpen, setAddContactModalOpen] =
useState(false);
  const [contacts, setContacts] = useState([]);
  const [selectedContacts, setSelectedContacts] = useState([]);
  const [contextMenu, setContextMenu] = useState({ visible: false, x:
0, y: 0, contactId: null });
  const [searchTerm, setSearchTerm] = useState('');
  const [refreshContacts, setRefreshContacts] = useState(false);
  const [isRegistering, setIsRegistering] = useState(false);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const [filteredContacts, setFilteredContacts] = useState([]);
  const { theme, language, setTheme, setLanguage } = useSettings();

  useEffect(() => {
    const checkAuth = () => {
      if (isTokenExpired()) {
        handleLogout();
      }
    };

    checkAuth();
    const interval = setInterval(checkAuth, 60000); // Проверка
каждую минуту

    return () => clearInterval(interval);
  }, []);

```

```

useEffect(() => {
  if (!isLoggedIn) return;

  const fetchSettings = async () => {
    try {
      setLoading(true);
      const token = getToken();
      if (isTokenExpired(token)) {
        handleLogout();
        return;
      }

      const response = await
fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/users`, {
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${token}`,
    'Content-Type': 'application/json',
  },
});

      if (response.status === 401) {
        handleLogout();
        return;
      }

      if (!response.ok) {
        throw new Error('Failed to fetch settings');
      }

      const settings = await response.json();
      setTheme(settings[0]?.theme || 'light');
      setLanguage(settings[0]?.language || 'en');
    } catch (error) {
      setError(error.message);
    } finally {
      setLoading(false);
    }
  };

  fetchSettings();
}, [isLoggedIn, setTheme, setLanguage]);

useEffect(() => {
  if (!isLoggedIn) return;

  const loadContacts = async () => {
    try {
      setLoading(true);
      const token = getToken();
      if (isTokenExpired(token)) {
        handleLogout();

```



```

        return;
    }

    const response = await
    fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/contacts`,
    {
        method: 'GET',
        headers: {
            'Authorization': `Bearer ${token}`,
            'Content-Type': 'application/json',
        },
    });

    if (response.status === 401) {
        handleLogout();
        return;
    }

    if (!response.ok) {
        throw new Error('Failed to fetch contacts');
    }

    const fetchedContacts = await response.json();
    console.log('Fetched contacts:', fetchedContacts);
    const currentUserId = getUserIdFromToken();
    const validContact = fetchedContacts.find(contact =>
    contact.contact.id !== currentUserId);

    if (validContact) {
        setActiveChat({ ...validContact.contact, type: 'contact'
    });
    }

    setContacts(fetchedContacts);
    } catch (error) {
        console.error('Error fetching contacts:', error);
    } finally {
        setLoading(false);
    }
    };

    loadContacts();
    }, [isLoggedIn, refreshContacts]);

    useEffect(() => {
        const loadContacts = async () => {
            if (searchTerm.length < 3) {
                setContacts([]);
                return;
            }
        }

        try {
            const token = getToken();

```

```

    const response = await
    fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/users?searchTerm=${encodeURIComponent(searchTerm)}`, {
      method: 'GET',
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json',
      },
    });

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const fetchedContacts = await response.json();
    console.log(fetchedContacts);
    setFilteredContacts(fetchedContacts);
  } catch (error) {
    console.error('Error fetching contacts:', error);
    setFilteredContacts([]);
  }
};

const debounceTimer = setTimeout(loadContacts, 300);
return () => clearTimeout(debounceTimer);
}, [searchTerm]);

const handleLoginSuccess = () => {
  setIsLoggedIn(true);
  setRefreshContacts(prev => !prev);
};

const handleLogout = () => {
  removeToken();
  setIsLoggedIn(false);
  setActiveChat(null);
  setError(null);
};

const handleSaveSettings = async (newTheme, newLanguage) => {
  try {
    setLoading(true);
    const token = getToken();
    if (isTokenExpired(token)) {
      handleLogout();
      return;
    }

    const userId = getUserIdFromToken();
    const response = await
    fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/users/${userId}`, {
      method: 'PATCH',

```

```

        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ theme: newTheme, language: newLanguage
    })),
  });

  if (response.status === 401) {
    handleLogout();
    return;
  }

  if (!response.ok) {
    const errorText = await response.text();
    throw new Error(`Failed to save settings: ${errorText}`);
  }

  setTheme(newTheme);
  setLanguage(newLanguage);
} catch (error) {
  setError(error.message);
} finally {
  setLoading(false);
}
};

const handleAddContact = async () => {
  if (!selectedContacts.length) return;

  try {
    setLoading(true);
    const token = getToken();
    if (isTokenExpired(token)) {
      handleLogout();
      return;
    }

    // Step 1: Fetch existing contacts
    const existingContactsResponse = await
fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/contacts`,
{
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${token}`,
    'Content-Type': 'application/json',
  },
});

    if (existingContactsResponse.status === 401) {
      handleLogout();
      setLoading(false); // Stop loading on logout
      return;
    }
  }

```

```

    }

    if (!existingContactsResponse.ok) {
      setLoading(false); // Stop loading on error
      throw new Error('Failed to fetch existing contacts');
    }

    const existingContacts = await existingContactsResponse.json();
    const existingContactIds = existingContacts.map(contact =>
contact.contact.id);

    const contactsToAdd = [];
    const alreadyExisting = [];
    const failedToAdd = []; // Для отслеживания ошибок при
добавлении новых

    // Step 2: Separate contacts to add and already existing
    selectedContacts.forEach(selectedId => {
      if (existingContactIds.includes(selectedId)) {
        const existingContact = filteredContacts.find(contact =>
contact.id === selectedId);
        if (existingContact) {
          alreadyExisting.push(existingContact.name);
        } else {
          alreadyExisting.push(`ID ${selectedId}`);
        }
      } else {
        contactsToAdd.push(selectedId);
      }
    });

    // Step 3: Inform the user if contacts already exist (optional,
but good UX)
    if (alreadyExisting.length > 0) {
      const message = `${language === 'en' ? 'The following users
were already in your contacts:' : 'Следующие пользователи уже были в
ваших контактах:'} ${alreadyExisting.join(', ')}.`;
      // We can set a message here, but continue to add the new
ones
      // setError(message); // Maybe combine messages later
      console.log(message); // Log or handle this message
    }

    // Step 4: Proceed only with contacts that are not already
existing
    if (contactsToAdd.length === 0) {
      // If no new contacts to add, just close modal and clear
state
      setAddContactModalOpen(false);
      setSelectedContacts([]);
      setSearchTerm('');
      setLoading(false);
      // If there were only existing contacts, show message now

```

```

        if (alreadyExisting.length > 0) {
            const message = `${language === 'en' ? 'All selected
users were already in your contacts.' : 'Все выбранные пользователи
уже были в ваших контактах.'}`;
            setError(message);
        }
        return;
    }

    // Step 5: Send POST requests for new contacts
    const promises = contactsToAdd.map(async (contactId) => {
        try {
            const response = await
fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/contacts`,
{
            method: 'POST',
            headers: {
                'Authorization': `Bearer ${token}`,
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ userId: contactId }),
        });

            // Проверяем статус ответа
            if (response.status === 409) {
                // Контакт уже существует - это не ошибка добавления, а
информационное сообщение
                console.log(`Contact with ID ${contactId} already
exists.`);
                // Можно добавить его в список "уже существующих" для
финального сообщения
                const existingContact = filteredContacts.find(contact =>
contact.id === contactId);
                if (existingContact &&
!alreadyExisting.includes(existingContact.name)) {
                    alreadyExisting.push(existingContact.name);
                } else if (!alreadyExisting.includes(`ID ${contactId}`))
{
                    alreadyExisting.push(`ID ${contactId}`);
                }
                return; // Продолжаем выполнение Promise.all, но без
обработки как ошибки
            }

            if (!response.ok) {
                const errorText = await response.text();
                console.error(`Failed to add contact ${contactId}:
${response.status} - ${errorText}`);
                failedToAdd.push(`ID ${contactId}`); // Отслеживаем
ошибки добавления
                // Не выбрасываем исключение здесь, чтобы Promise.all
завершился для всех запросов
            } else {

```

```

        const addedContact = await response.json();
        console.log('Added contact:', addedContact); // Лог
успешного добавления
        // Возможно, здесь потребуется обновить локальное
состояние контактов или полагаться на refreshContacts
    }
    } catch (error) {
        console.error('Error during POST for contact', contactId,
error);
        failedToAdd.push(`ID ${contactId}`); // Отслеживаем ошибки
запроса
    }
    });

    await Promise.all(promises); // Ожидаем завершения всех POST
запросов

    // Step 6: Provide feedback based on results
    let feedbackMessage = '';
    if (alreadyExisting.length > 0) {
        feedbackMessage += `${language === 'en' ? 'Some users were
already in your contacts:' : 'Некоторые пользователи уже были в ваших
контактах:'} ${alreadyExisting.join(', ')} `;
    }
    if (failedToAdd.length > 0) {
        feedbackMessage += `${language === 'en' ? 'Failed to add:'
: 'Не удалось добавить:'} ${failedToAdd.join(', ')} `;
    }
    if (alreadyExisting.length === 0 && failedToAdd.length === 0)
{
        feedbackMessage = `${language === 'en' ? 'Contacts added
successfully.' : 'Контакты успешно добавлены.'}`;
    } else if (contactsToAdd.length > 0 && failedToAdd.length ===
0 && alreadyExisting.length === 0) {
        // All selected and new were added successfully
        feedbackMessage = `${language === 'en' ? 'Contacts added
successfully.' : 'Контакты успешно добавлены.'}`;
    } else if (contactsToAdd.length > 0 && failedToAdd.length ===
0 && alreadyExisting.length > 0) {
        // Some were new and added, some already existed
        feedbackMessage += `${language === 'en' ? 'New contacts
added successfully.' : 'Новые контакты успешно добавлены.'}`;
    }
    if (feedbackMessage) {
        setError(feedbackMessage); // Use error state to show
combined feedback
    }
    // Close modal and clear state after processing
    setAddContactModalOpen(false);
    setSelectedContacts([]);
    setRefreshContacts(prev => !prev); // Обновление списка
контактов
    setSearchTerm('');

```

```

    } catch (error) {
      console.error('Overall error adding contact:', error);
      setError(error.message); // Show general error message
    } finally {
      setLoading(false); // Ensure loading is stopped
    }
  };

const handleDeleteContact = async (contactId) => {
  try {
    setLoading(true);
    const token = getToken();
    if (isTokenExpired(token)) {
      handleLogout();
      return;
    }

    const response = await fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/contacts`,
    {
      method: 'DELETE',
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ userId: contactId }),
    });

    if (response.status === 401) {
      handleLogout();
      return;
    }

    if (response.ok) {
      setRefreshContacts(prev => !prev);
      setContextMenu({ visible: false, x: 0, y: 0, contactId: null
    });
    }
  } catch (error) {
    setError(error.message);
  } finally {
    setLoading(false);
  }
};

const handleRegisterSuccess = () => {
  handleLoginSuccess();
};

const handleSearchChange = (e) => {
  const value = e.target.value;
  setSearchTerm(value);
};

```

```

};

return (
  <SettingsContext.Provider value={{ theme, setTheme, language,
setLanguage }}>
    {isLoggedIn ? (
      <div className={`app-container ${theme}`}>
        <div className="sidebar" style={{ backgroundColor: theme
=== 'dark' ? '#333' : '#fff' }}>
          <div className="tabs">
            <button
              className={activeTab === 'contacts' ? 'active' : ''}
              onClick={() => setActiveTab('contacts')}>
              {language === 'en' ? 'Contacts' : 'Контакты'}
            </button>
            <button
              className={activeTab === 'channels' ? 'active' : ''}
              onClick={() => setActiveTab('channels')}>
              {language === 'en' ? 'Channels' : 'Каналы'}
            </button>
          </div>
          <div
            className="list-container"
            style={{
backgroundColor: theme === 'dark' ? '#444' : '#f9f9f9' }}>
            {activeTab === 'contacts' ? (
              <>
                <ContactsList
                  onSelect={(chat) => setActiveChat({ ...chat,
type: 'contact' })}
                  refreshContacts={refreshContacts}
                  onContextMenu={(e, contactId) => {
                    e.preventDefault();
                    setContextMenu({
                      visible: true,
                      x: e.clientX,
                      y: e.clientY,
                      contactId: contactId,
                    });
                  }}
                </>
              </div>
              {contextMenu.visible && (
                <ContextMenu
                  x={contextMenu.x}
                  y={contextMenu.y}
                  onDelete={() =>
handleDeleteContact(contextMenu.contactId)}
                  theme={theme}
                  language={language}
                  resource='contact'
                </>
              )}
            <div
              style={{ display: contextMenu.visible ? 'block'
: 'none' }}

```



```

        onClick={() => setContextMenu({ visible: false,
x: 0, y: 0, contactId: null })}
        className="context-menu-overlay"
      />
    </>
  ) : (
    <ChannelsList onSelect={({ chat } => setActiveChat({
...chat, type: 'channel' })} />
  )}
</div>
<div className="profile-buttons" style={{
backgroundColor: theme === 'dark' ? '#444' : '#f9f9f9' }}>
  <button className="profile-button" onClick={() =>
setProfileOpen(true)}>
    {language === 'en' ? 'Profile' : 'Профиль'}
  </button>
  <button className="add-contact-button" onClick={() =>
setAddContactModalOpen(true)}>
    {language === 'en' ? 'Add Contact' : 'Добавить
контакт'}
  </button>
</div>

<div className="main" style={{ backgroundColor: theme ===
'dark' ? '#222' : '#fff' }}>
  {activeChat ? (
    <ChatWindow chat={activeChat} theme={theme}
language={language} />
  ) : (
    <div className="placeholder">
      {language === 'en' ? 'Select a contact or channel to
start chatting.' : 'Выберите контакт или канал для начала чата.'}
    </div>
  )}
</div>

{isProfileOpen && (
  <ProfileSettings
    onClose={() => setProfileOpen(false)}
    onSave={handleSaveSettings}
    currentTheme={theme}
    currentLanguage={language}
  >
    <button className="logout-button"
onClick={handleLogout}>
      {language === 'en' ? 'Log Out' : 'Выйти'}
    </button>
  </ProfileSettings>
)}

{isAddContactModalOpen && (
  <div className="modal">

```

```

        <div className="modal-content" style={{ background:
theme === 'dark' ? 'var(--dark-modal-bg)' : 'var(--modal-bg)' }}>
        <span className="close" onClick={() =>
setAddContactModalOpen(false)}>&times;</span>
        <h2>{language === 'en' ? 'Add Contact' : 'Добавить
контакт'}</h2>
        <input
            type="text"
            placeholder={language === 'en' ? 'Search
contacts...' : 'Поиск контактов...'}
            value={searchTerm}
            onChange={handleSearchChange}
        />
        <ul>
            {filteredContacts.map(contact => (
                <li key={contact.id}>
                    <input
                        type="checkbox"
                        value={contact.id}
                        onChange={(e) => {
                            const id = parseInt(e.target.value);
                            setSelectedContacts(prev =>
                                e.target.checked
                                    ? [...prev, id]
                                    : prev.filter(contactId => contactId
!= id)
                                )};
                        }}
                    />
                    {contact.name}
                </li>
            ))}
        </ul>
        <button onClick={handleAddContact}>
            {language === 'en' ? 'Add Selected' : 'Добавить
выбранные'}
        </button>
    </div>
</div>
)}
</div>
) : (
    <div className="auth-page-container">
        <div className="auth-wrapper">
            {isRegistering ? (
                <Register
                    onRegisterSuccess={handleRegisterSuccess}
                    switchToLogin={() => setIsRegistering(false)}
                />
            ) : (
                <Login
                    onLoginSuccess={handleLoginSuccess}
                    switchToRegister={() => setIsRegistering(true)}

```

```

        />
      })
    </div>
  </div>
})

{loading && (
  <div className="loading-overlay">
    <div className="loading-spinner"></div>
  </div>
)}
{error && (
  <div className="error-modal">
    <div className="error-content">
      <span      className="close-error"      onClick={() =>
setError(null)}>&times;</span>
      <p>{error}</p>
      <button onClick={() => setError(null)}>OK</button>
    </div>
  </div>
)}
</SettingsContext.Provider>
);
}

// Компонент для управления настройками
const SettingsProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  const [language, setLanguage] = useState('en');

  return (
    <SettingsContext.Provider value={{ theme, language, setTheme,
setLanguage }}>
      {children}
    </SettingsContext.Provider>
  );
};

export default () => (
  <SettingsProvider>
    <App />
  </SettingsProvider>
);

```

Листинг – Код компонента App

## Приложение Г

```
import React, { useState } from 'react';

const Login = ({ onLoginSuccess, switchToRegister }) => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const [isLoading, setIsLoading] = useState(false);

  const handleLogin = async (e) => {
    e.preventDefault();
    setError('');
    setIsLoading(true);

    try {
      const response = await fetch(`https://${process.env.REACT_APP_USER_CLIENT_HOST}/auth/login`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ email: username, password }),
      });

      if (!response.ok) {
        throw new Error('Login failed');
      }

      const data = await response.json();
      localStorage.setItem('accessToken', data.token);
      onLoginSuccess();
    } catch (error) {
      console.error('Error during login:', error);
      setError('Login failed. Please check your credentials.');
```

finally {  
 setIsLoading(false);  
}

```
};

return (
  <div className="auth-container">
    <div className="auth-card">
      <h2 className="auth-title">Welcome Back</h2>
      <p className="auth-subtitle">Sign in to continue</p>

      {error && <div className="auth-error">{error}</div>}

      <form onSubmit={handleLogin} className="auth-form">
        <div className="form-group">
          <label htmlFor="login-email">Email</label>
          <input
```

```

        id="login-email"
        type="text"
        placeholder="Enter your email"
        value={username}
        onChange={ (e) => setUsername(e.target.value) }
        required
      />
    </div>

    <div className="form-group">
      <label htmlFor="login-password">Password</label>
      <input
        id="login-password"
        type="password"
        placeholder="Enter your password"
        value={password}
        onChange={ (e) => setPassword(e.target.value) }
        required
      />
    </div>

    <button
      type="submit"
      className="auth-button"
      disabled={isLoading}
    >
      {isLoading ? 'Signing in...' : 'Sign In'}
    </button>
  </form>

  <div className="auth-footer">
    Don't have an account?{' '}
    <span className="auth-link" onClick={switchToRegister}>
      Sign up
    </span>
  </div>
</div>
</div>
);
};

export default Login;

```

Листинг – Код компонента страницы Login